

Г. МАЙЕРС

АРХИТЕКТУРА
СОВРЕМЕННЫХ

ЭВМ





АРХИТЕКТУРА СОВРЕМЕННЫХ ЭВМ

ADVANCES IN COMPUTER ARCHITECTURE

SECOND EDITION

GLENFORD J. MYERS

INTEL CORPORATION
SANTA CLARA, CALIFORNIA

A WILEY-INTERSCIENCE PUBLICATION
JOHN WILEY & SONS
NEW YORK CHICHESTER BRISBANE TORONTO SINGAPORE
1982

Г. МАЙЕРС

АРХИТЕКТУРА ЭВМ СОВРЕМЕННЫХ

1

В 2-х КНИГАХ

ПЕРЕВОД С АНГЛИЙСКОГО
ПОД РЕДАКЦИЕЙ КАНД. ТЕХН. НАУК
В. К. ПОТОЦКОГО



МОСКВА «МИР» 1985

ББК 32.973
М 14
УДК 681.3

Майерс Г.

М 14 Архитектура современных ЭВМ: В 2-кн. Кн. 1.
Пер. с англ. — М.: Мир, 1985. — 364 с., ил.

Монография известного американского специалиста в области вычислительной техники посвящена анализу фундаментальных проблем архитектуры современных вычислительных машин; описывается ряд достижений на пути преодоления этих проблем. Выходит на русском языке в двух книгах. В книге 1 дан критический анализ архитектуры традиционных ЭВМ модели фон Неймана, формулируются основные требования к архитектуре машин ближайшего будущего, описываются решения некоторых задач на примере вычислительных систем с архитектурой, ориентированной на языки программирования (системы SPLM, SYMBOL, BI700) и программное обеспечение (система SWARD).

Для специалистов в области вычислительной техники и программирования. Может быть полезна студентам старших курсов соответствующих специальностей вузов.

М 2405000000-242
041(01)-85 165-85, ч. 1

ББК 32.973
6Ф7.3.

Редакция литературы по информатике и электронике

Copyright © 1978 by John Wiley & Sons, Inc. All rights reserved. Authorized translation from English language edition published by John Wiley & Sons, Inc.

© Перевод на русский язык, «Мир», 1985

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

На протяжении трех последних десятилетий вычислительная техника все стремительнее и шире охватывает различные сферы человеческой деятельности. Эта тенденция наблюдается и в настоящее время. Существует много причин столь бурного развития вычислительной техники, однако доминирующая роль принадлежит достижениям в области технологии производства электронных схем: широкое распространение больших, а в настоящее время и сверхбольших интегральных схем, массовое производство семейств микропроцессоров. Значительное увеличение производительности упомянутых схем, являющихся аппаратной основой ЭВМ, и существенное снижение их стоимости, обусловленные не только техническим прогрессом, но и массовым характером их производства, явились предпосылкой для пересмотра принципов организации ЭВМ, базирующихся главным образом на традиционной модели вычислительной машины фон Неймана, разработанной еще в 40-х годах.

Автор предлагаемой читателю книги определяет необходимость указанного пересмотра основ организации ЭВМ наличием так называемого семантического разрыва между возможностями аппаратных средств современных ЭВМ, с одной стороны, и программного обеспечения этих машин, с другой. Рабочие характеристики аппаратных средств и сейчас, как правило, обуславливаются основополагающими принципами модели фон Неймана, тогда как параметры программного обеспечения во многом зависят от языков программирования, а следовательно, и от класса задач, подлежащих решению. Определяя проблемы, стоящие перед разработчиком архитектуры современной ЭВМ, прежде всего как проблемы организации интерфейса между требованиями, предъявляемыми к программному обеспечению, и возможностями, предоставляемыми современными аппаратными средствами, автор проводит разносторонний и, как правило, глубокий сравнительный анализ принципов организации целого ряда вычислительных систем, появившихся в последние годы и являющихся отступлением, а иногда и отказом от традиционной модели машины фон Неймана. Читатель найдет много интересного материала, посвященного описанию не только ЭВМ нетрадиционной архитектуры (SWARD, IAPX 432), ориентированных на максимальное согласование требований новых языков программирования (Паскаль, Ада), быстро завоевывающих умы и сердца программистов во всем мире, но и так называемых машин баз данных (RAP, CASSM) и потоков данных (модель Массачусетского технологического института). На протяжении всей книги красной нитью проходят два важных положения современного подхода к проблемам организации вычислительного процесса: 1) предметом вычислений следует считать не данные, хранящиеся в ячейках памяти, а разнообразные классы абстрактных объектов, корректная манипуляция которыми и является

целью организации вычислений; 2) основные характеристики вычислительной системы должны оцениваться не скоростью выполнения отдельных элементарных операций, а машинным временем и объемом памяти, необходимыми для выполнения операций над абстрактными объектами в пределах строго определенных контекстов программ.

Книга представляет собой одну из первых попыток всестороннего и систематизированного исследования проблем современных ЭВМ с позиций известного противоречия между их аппаратными средствами и программным обеспечением. Критика традиционных решений сопровождается анализом современных достижений на пути совершенствования архитектуры вычислительных систем. Все это делает данную книгу крайне полезной для лиц, специализирующихся в области разработки средств вычислительной техники, роль которой в окружающем нас мире трудно переоценить.

Перевод книги выполнен канд. техн. наук В. И. Гуревичем (гл. 1—12, 17), канд. физ.-мат. наук Ю. Н. Жежелем (гл. 13—16, 18—21) и М. Л. Мирным (гл. 22—24).

В. К. Потоцкий

ПРЕДИСЛОВИЕ КО ВТОРОМУ ИЗДАНИЮ

За три года, прошедших со времени первого издания данной книги, достигнуты значительные успехи по совершенствованию архитектуры вычислительных машин. Этому способствовал целый ряд факторов, в том числе появление промышленно выпускаемых сверхбольших схем, возобновление интереса к разработке языков программирования и признание широким кругом специалистов в области вычислительной техники тех ее основных принципов и проблем, которым посвящено первое издание настоящей книги. (Одной из таких проблем является разработка программного обеспечения вычислительной системы путем совершенствования архитектуры ее аппаратных средств.) Упомянутые факторы нашли свое воплощение в таких вычислительных системах, как iAPX 432 фирмы Intel, система 38 фирмы IBM, а также новый вариант машины SWARD (первоначальный вариант этой машины описан в первом издании книги).

Побудительными мотивами к выпуску второго издания послужили не только разработка упомянутых систем, но и появление таких новых понятий в области вычислительной техники, как машины потоков данных и машины баз данных. Все это позволило расширить круг вопросов, освещенных в первом издании, а также более детально проанализировать их с учетом личного опыта автора, накопленного в процессе реализации аппаратных средств и программного обеспечения системы.

По структуре второе издание мало отличается от первого. Часть I (гл. 1—4) существенно переработана, содержит теперь больше количественных оценок анализируемых вопросов и, как хотелось бы надеяться, является более аргументированной. Так, в гл. 4 в качестве примеров приводятся сведения, заимствованные из документации систем 68000 фирмы Motorola и 38 фирмы IBM. Примеры, используемые в первом издании в качестве предмета анализа в частях II, III и IV, сохраняются и во втором издании, причем удалось устранить ряд ошибок и неточностей, допущенных в предыдущем издании. Часть V, посвященная принципам системы SWARD, полностью переработана, благодаря чему отражает текущее состояние достижений в проектировании архитектуры вычислительных систем этого типа. Часть VI является новой и содержит описание системы iAPX 432 фирмы Intel, т. е. системы, во многом изменявшей существующие представления об архитектуре микропроцессоров. Часть VII представляет собой расширенный вариант изложения материала первого издания книги, посвященного машинам баз данных. Новым по сравнению с первым изданием является здесь описание машин потоков данных. Завершающая книгу часть VIII включает некоторые общие сведения об особенностях разработки архитектуры вычислительной системы.

Я хотел бы выразить благодарность Джорджу Коксу, Джастину Ретнеру и Кену Оперлю — сотрудникам фирмы Intel, а также Дейву Дятцелу (Bell Laboratories), которые внесли определенный вклад в подготовку второго издания книги.

Санта-Клара, Калифорния

октябрь 1981

Гленфорд Дж. Майеро

ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ

Начиная с 50-х годов мы являемся свидетелями многих достижений в создании и совершенствовании вычислительных систем. Громадные успехи получены в области программного обеспечения ЭВМ, благодаря чему в настоящее время стали эффективнее средства программного обеспечения, методология проектирования и языки программирования. Наборы прикладных программ отличаются более сложной структурой и более широкими возможностями. Появились алгоритмы, построенные на новых принципах. Изменились приемы и средства программирования, так что стали ясными принципы построения таких программ, как компиляторы и программы операционной системы. Большой прогресс достигнут также в области аппаратных средств ЭВМ: на несколько порядков увеличилось быстродействие и плотность монтажа электронных схем; разработаны запоминающие устройства на новой технологической базе; предложены более совершенные алгоритмы функционирования аппаратных средств; широко реализуются принципы микропрограммирования. Однако происшедшие изменения не сопровождались какими-либо значительными достижениями в отношении улучшения аппаратно-программного интерфейса вычислительной системы — того уровня ее организации, который получил наименование *архитектуры ЭВМ*. Чтобы быть объективным в оценке развития вычислительной техники, следует отметить, что имели место некоторые и весьма существенные успехи в развитии архитектуры ЭВМ, однако они не привлекли к себе достаточного внимания специалистов и не получили адекватного воплощения в большинстве традиционных вычислительных систем. Подтверждением этого может служить тот факт, что система команд многих современных больших систем, мини- и микро-ЭВМ до сих пор удивительно похожа на систему команд аналогичных ЭВМ, сконструированных еще в 50-е годы.

Сходство архитектур современных и более ранних вычислительных систем может вызывать даже чувство удовлетворения: с одной стороны, имеются громадные достижения в области аппаратных и программных средств ЭВМ, с другой — существует преемственность в архитектуре машины различных поколений. Это позволяет сделать вывод, что в 40—50-х годах все принципиальные изобретения в области архитектуры ЭВМ уже были сделаны, и, следовательно, архитектура вычислительных систем будущего не претерпит никаких изменений. Именно такое отношение к проблемам архитектуры ЭВМ побудило автора написать эту книгу. В данной работе делается попытка поставить под сомнение возникшее чувство удовлетворения преемственностью архитектуры машины прошлого и настоящего, показать, что архитектуре современных ЭВМ присущ ряд серьезных проблем, а также обсудить новые направления совершенствования архитектуры, которые могут привести к решению указанных проблем.

Назначение этой книги хорошо определяет и тот выбор, который пришлось делать автору между двумя первоначальными вариантами ее названия. Первое название, к которому склонялся автор, — «Архитектура ЭВМ пятого поколения». Поколение машин предполагалось назвать пятым, потому что, по мнению автора, четвертое поколение вычислительных машин уже находилось на чертежных листах проектировщиков ЭВМ, причем эти системы, безусловно, сохраняли архитектуру ранее созданных машин. Однако от такого названия пришлось отказаться, поскольку оно представлялось слишком вызывающим и, кроме того, в определенном смысле дезинформировало бы читателя. Это связано с тем, что целый ряд положений, рассматриваемых в данной книге, присущ и некоторым ЭВМ второго поколения. Второе название из первоначально предполагаемых — «Архитектура ЭВМ второй эры» — было также отвергнуто, поскольку читатель мог бы легко принять термин «эра» за термин «поколение», поэтому у него могло создаться впечатление, что эта книга — исторический обзор таких вычислительных машин второго поколения, как IBM 1401, Burroughs 200 и других машин, построенных на основе транзисторов.

Структура данной книги такова, что все главы объединены в шесть частей. В части I даются определение архитектуры вычислительной машины, критическая оценка архитектуры современных ЭВМ и формулировка набора характеристик, которыми должна обладать архитектура ЭВМ будущего. В частях II—V описываются отдельные вычислительные системы, воплощающие в себе современные достижения в развитии архитектуры ЭВМ. Эти системы иллюстрируют также реализацию ряда характеристик архитектуры, целесообразность которых определена в части I. В части VI анализируются некоторые проблемы, связанные с архитектурой ЭВМ: организация ввода-вывода, оптимизация, или «настройка», архитектуры конкретной вычислительной системы и др.

Книга предназначена для читателей двух категорий: ее могут использовать студенты, специализирующиеся в области вычислительной техники, как вторую часть учебного пособия по архитектуре ЭВМ. (Предполагается, что первая часть пособия знакомит с основами архитектуры традиционных ЭВМ.) Кроме того, книга представляет интерес для специалистов, так как расширяет существующие представления в области вычислительной техники. Книга рассчитана на подготовленного читателя. Он должен быть знаком с основами вычислительных систем, в том числе с языками программирования высокого уровня, машинным языком или языком ассемблера традиционной ЭВМ, принципами организации и функционирования операционной системы и компиляторов, а также микропрограммированием. Автор считает, что знание перечисленных вопросов является необходимой предпосылкой для разработки архитектуры ЭВМ.

По глубокому убеждению автора, наиболее эффективный способ изучения архитектуры ЭВМ состоит в использовании описания архитектуры конкретной ЭВМ и мысленном компилировании программ, написанных на языке высокого уровня, применительно к этой системе. Именно так построены многие примеры, приводимые в книге. При пользовании книгой как учебным пособием рекомендуется воспроизвести в уме процедуру компилирования небольших программ на языках ПЛ/1, КОБОЛ или ФОРТРАН, ориентируясь на архитектуру каждой из рассматриваемых вычислительных систем.

Наконец, следует отметить, что излагаемый в книге материал отображает точку зрения автора, которая не обязательно совпадает с мнением фирмы IBM, и знакомит с направлениями дальнейшего совершенствования ЭВМ, выпускаемых этой фирмой.

ЧАСТЬ I

НЕОБХОДИМОСТЬ УСОВЕРШЕНСТВОВАНИЯ АРХИТЕКТУРЫ ЭВМ

ГЛАВА I

ОПРЕДЕЛЕНИЕ ПОНЯТИЯ «АРХИТЕКТУРА ЭВМ»

Поскольку термин «архитектура ЭВМ» трактуется самым различным образом, необходимо точно определить, что под ним будет подразумеваться в данной книге. С этой целью рассмотрим процесс проектирования вычислительной системы в целом.

Для неспециалиста термин «архитектура» обычно ассоциируется со строительными объектами, и действительно имеется много общего между таким толкованием этого термина и понятием «архитектура ЭВМ». Однако существует неправильное мнение, что архитектура зданий находит свое воплощение в их внешнем виде, высоте холла, размещении зеркал, интерьере.

В действительности указанные факторы хотя и играют определенную роль, но представляют лишь небольшую часть проблемы. Усилия архитектора в равной мере должны быть направлены на то, чтобы здание соответствовало своему функциональному назначению, т. е. удовлетворяло требованиям, предъявляемым к нему лицами, эксплуатирующими здание. Это относится как к жилым помещениям, так и к служебным, например транспортным сооружениям, учреждениям связи. Здание должно быть удобным в эксплуатации (близость подсобных помещений к основным служебным, наличие системы отопления), безопасным (наличие системы противопожарной сигнализации, аварийных выходов, системы охраны персонала), надежным (обеспечение требуемого запаса прочности, использование испытанной на практике технологии сооружения зданий). Наряду с этим необходимо учитывать экономические показатели и ряд других факторов. Таким образом, архитектор имеет дело с проблемами, лежащими по обе стороны границы взаимодействия проектируемого сооружения с лицами, его эксплуатирующими, и окружающей средой. Более того, на него возлагается задача формального определения этих границ, т. е. контуров здания и всех его подсистем. Однако, когда появляются такие требования к проекту системы (в частности, к сооружению зда-

ния), как использование современной технологии, обеспечение удобства в эксплуатации, безопасности, надежности, приемлемой стоимости, становится ясно, что речь идет о решении технических проблем и профессии инженера. Вот почему можно сказать, что архитектура — это одна из форм инженерии, и читатель должен постоянно помнить, что решения архитектурных

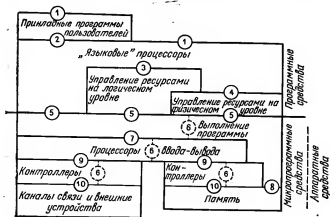


Рис. 1.1. Многоуровневая организация архитектуры вычислительной системы.

задач следует ожидать скорее от инженеров, чем от представителей мира искусства.

Применительно к вычислительным системам термин «архитектура» может быть определен как распределение функций, реализуемых системой, по отдельным ее уровням и точное определение границ между этими уровнями. Таким образом, если архитектуре системы отведен некоторый уровень, то в первую очередь необходимо установить, какие системные функции полностью или частично выполняются компонентами системы, находящимися выше и ниже заданного уровня. После решения этой задачи следующий шаг заключается в точном определении интерфейсов для рассматриваемого уровня.

Таким образом, архитектура вычислительной системы предполагает многоуровневую организацию. Архитектуру вычислительной системы можно определить путем выявления ее отличий от других видов архитектуры. Так, специфическим свойством архитектуры вычислительной системы является возможность выделения в ней набора уровней абстракции (рис. 1.1). (При рассмотрении проблем, связанных с отображением некоторой концептуальной модели, подобной изображенной на

рис. 1.1, может сложиться впечатление, что требуемые технические решения уже определены — в действительности это не так!)

Архитектура первого уровня, называемая *архитектурой системы* и символически помеченная цифрой 1, определяет, какие функции по обработке данных выполняются системой в целом, а какие возлагаются на «внешний мир» (пользователей, операторов ЭВМ, администраторов баз данных и т. п.). Система взаимодействует с внешним миром через два набора интерфейсов: языкн (такие, как язык оператора терминала, языки программирования, язык описания и манипулирования базой данных, язык оператора ЭВМ, язык управления заданиями) и системные программы (прикладные программы, созданные разработчиком системы, например программы-утилиты, программы редактирования, сортировки, восстановления и обновления информации). Разработка архитектуры системы предполагает определение обеих групп этих интерфейсов.

Интерфейсы 2, 3 и 4 разграничивают определенные уровни внутри программного обеспечения, хотя они в большинстве случаев не имеют каких-либо общепризнанных наименований. Если программы, реализующие прикладные задачи, написаны на языках программирования, не входящих в число тех, которые предоставлены в распоряжение пользователя, то можно говорить об архитектуре уровня, назначение которого — определение указанных языков. Трансляторы таких языков в свою очередь взаимодействуют с более низкими уровнями программного обеспечения, обозначенными на абстрактной модели архитектуры как уровни 3 и 4. Уровень управления логическими ресурсами может включать реализацию таких функций, как управление базой данных, файлами, виртуальной памятью, сетевой телеобработкой. К уровню управления физическими ресурсами относятся функции управления внешней и оперативной памятью, внутренними процессами, протекающими в системе (например, процессами планирования и синхронизации), а также другими аппаратными средствами. Из-за отсутствия лучшего термина о всех трех уровнях 2—4 будем говорить как об *архитектуре программного обеспечения*.

Уровень 5 отражает основную линию разграничения системы, а именно границу между системным программным¹⁾ и аппаратным обеспечением (термин «аппаратное обеспечение» используется здесь для обозначения как микропрограмм, так и электронных логических схем). Термины «микропрограммное управление/микрокод/микропрограмма» не имеют стандартного универсального определения и часто используются непра-

¹⁾ То есть операционной системой ЭВМ. — *Прим. перев.*

вильно. В данной книге они употребляются в традиционном смысле: *микропрограмма* — это записанная в памяти программа, которая фактически управляет передачей всех символов и данных в физических компонентах системы, таких, как шины, регистры, сумматоры или процессор; другим альтернативным средством управления передачей сигналов и данных является использование последовательностных логических схем [1]. Итак, уровень (интерфейс) 5 позволяет представлять физическую структуру системы абстрактно независимо от способа реализации. Разграничение функций, выполняемых выше и ниже этого уровня, и определение интерфейса 5 — одна из составных частей процесса разработки *архитектуры ЭВМ*.

Эту идею можно развить дальше и говорить о распределении функций между отдельными частями физической системы. Например, интерфейс 7 определяет, какие функции реализуют центральные процессоры (ЦП), выполняющие программы, а какие процессоры ввода-вывода (т. е. каналы).

Архитектура другого уровня определяет разграничение функций между процессорами ввода-вывода и контроллерами (устройствами управления) внешних устройств. В свою очередь можно разграничить функции, реализуемые контроллерами и самими устройствами ввода-вывода (терминалами, модемами, накопителями на магнитных дисках и магнитных лентах). Архитектура таких уровней (интерфейсы 7, 9 и 10) может быть названа *архитектурой физического ввода-вывода*.

Осталась нерассмотренной архитектура уровней 8 (интерфейс между процессором и основной памятью) и 6. Функции каждого процессора и контроллера внешнего устройства могут быть распределены между микропрограммами и комбинационными и последовательностными логическими схемами. Следовательно, интерфейс 6 представляет собой интерфейс микропрограммы (т. е. обеспечивает согласование потока данных и управляющих сигналов с форматом микрокоманд) внутри каждого процессора. Архитектура уровня 6 может быть названа *архитектурой микропрограммного управления*. Архитектуру уровней 6 и 8 также часто называют *архитектурой процессора* или *организацией процессора*.

Последняя разновидность архитектуры, в явном виде не показанная на рис. 1.1, может быть определена как *мультипроцессорная архитектура* (она не представлена на рисунке, поскольку отражает скорее вертикальный, чем горизонтальный разрез системы). Такая архитектура предусматривает распределение функций между группой процессоров (например, когда вычислительная система должна содержать периферийный процессор для управления базой данных или когда при использовании системы, построенной на микропроцессорах Intel 8086, на

одном кристалле реализуется арифметическое устройство с развитой логикой типа 8087) с разграничением функций вычислительной системы между этими процессорами и определенным соответствующим интерфейсом.

Приведенный здесь общий обзор архитектуры различных уровней организации вычислительной системы позволяет теперь определить термин «архитектура ЭВМ». *Архитектура ЭВМ* — это абстрактное представление или определение физической системы (микропрограммы и комплекса аппаратных средств) с точки зрения программиста, разрабатывающего программы на машинно-ориентированном языке, или разработчика компилятора. Она определяет принципы организации вычислительной системы и функции процессора и не отражает такие проблемы, как управление и передача данных внутри процессора, конструктивные особенности логических схем и специфика технологии их производства. Иногда (например, в [2]) некоторые или все перечисленные здесь проблемы входят в определение понятия «архитектура ЭВМ», однако в данной книге подход иной, поскольку они представляют отдельные и достаточно точно определенные технические вопросы.

Таким образом, архитектор ЭВМ должен принять решение по трем обширным группам проблем: определить форму представления программы для машины и правила ее интерпретации этой машиной; установить способы адресации данных в этих программах; задать форму представления данных. При решении каждой из перечисленных групп проблем разработчик архитектуры ЭВМ сталкивается с такими задачами, как определение минимально адресуемой области памяти, типов и форматов данных, кодов операций и форматов машинных команд, способов адресации и защиты памяти, механизма управления последовательностью выполнения команд, интерфейса машины с устройствами ввода-вывода.

В качестве примера, поясняющего различие между архитектурой ЭВМ и архитектурой ее отдельных уровней, можно рассмотреть семейство ЭВМ фирмы IBM, получившее название Система 360/370, которое разрабатывалось с целью создания ряда процессоров, «вписывающихся» в общую архитектуру ЭВМ, но имеющих различную внутреннюю структуру для обеспечения возможности выбора определенных соотношений между стоимостью и производительностью вычислительной системы. За исключением двух вопросов, техническая документация «Принципы работы Системы 370» [3] определяет архитектуру ЭВМ для всех процессоров данного семейства. Первое различие касается архитектуры процессора — способа, которым процессор «информирует» программное обеспечение о происшедшем машинном сбое. Этот вопрос излагается в отдельных руковод-

ствах для каждого процессора. Второе различие, выявляющееся при описании работы устройств ввода-вывода, заключается в отсутствии в упомянутой документации определения командных слов канала (терминология Системы 370). Командные слова канала определены в отдельных руководствах по эксплуатации каждого устройства ввода-вывода.

РОЛЬ РАЗРАБОТЧИКА АРХИТЕКТУРЫ ЭВМ

После того как определено понятие «архитектура ЭВМ», необходимо выяснить, какая работа возлагается на архитектора — разработчика ЭВМ. Согласно изложенному выше, разработка архитектуры ЭВМ сводится в основном к установлению границ между отдельными уровнями вычислительной системы с многоуровневой организацией. После предварительного определения функций, выполняемых системой в целом, задача архитектора ЭВМ состоит в распределении функций системы между отдельными уровнями. Используя набор таких критериев, как стоимость, быстродействие, надежность (зачастую эти критерии противоречивы), разработчик выясняет, какие функции или части функций реализуются ниже установленной границы, а какие выше. После этого требуется точно указать саму границу (т. е. «очертить» пределы действия архитектуры определенного уровня).

Процесс разработки архитектуры ЭВМ может быть структурно представлен таким же образом, как и большинство других процессов проектирования системы [4]. В идеальном случае разработчик архитектуры ЭВМ должен решать свою задачу в следующей последовательности: 1) анализ требований, предъявляемых к системе; 2) составление спецификаций; 3) изучение известных решений; 4) разработка функциональной схемы; 5) разработка структурной схемы; 6) отладка проекта; 7) оценка проекта.

Этап анализа требований состоит в основном в определении общей архитектуры системы с выделением факторов, несущих основную функциональную нагрузку. Подобными факторами являются количество и специфика требуемых языков программирования, характеристики периферийных устройств, способ и характер взаимодействия с окружающей средой (например, режим реального времени, телеобработки данных, разделения времени, невозможность несанкционированного доступа к системе, коммерческий или научный характер работ, подлежащих выполнению на ЭВМ), требования к операционной системе и т. д.

Анализ требований, предъявляемых к проектируемой системе, включает также технико-экономические вопросы, связанные

с дальнейшей эксплуатацией системы, и вопросы сбыта. Одним из важных экономических вопросов является определение количества систем, которые должны быть изготовлены, поскольку этот показатель оказывает существенное влияние на соотношение между стоимостью аппаратного и программного обеспечения. (Производство программного обеспечения не сопровождается затратами на изготовление дополнительного оборудования, если не считать потребность в запоминающих устройствах для программ; в то же время производство аппаратных средств ЭВМ сопряжено с определенными материальными затратами.) Типичной проблемой сбыта является соблюдение требования совместимости на определенном уровне (например, на уровне языка программирования) разработанной вычислительной системы и аналогичной системы, находящейся в эксплуатации.

Составление перечня требований и спецификаций параметров проектируемой системы включает формулировку используемых критериев и оценку их значимости, определение функций системы в соответствии с этими критериями, выявление других факторов, таких, как ограничения, накладываемые на параметры системы (это обусловлено возможностями имеющихся аппаратных средств или разработанной технологией изготовления требуемых полупроводниковых элементов и схем). Используемые критерии включают стоимость проектируемой системы, ее надежность, трудоемкость реализации, способность к расширению возможностей, совместимость с системами подобного типа, быстродействие, защиту от несанкционированного доступа, гибкость, степень риска успеха проекта в целом, простоту программирования. Поскольку некоторые из перечисленных критериев противостоят друг другу, их следует рассматривать с учетом соответствующих приоритетов.

О необходимости изучения существующих систем и решений можно было бы и не упоминать, если бы не столь частое пренебрежение этим столь важным этапом процесса разработки ЭВМ.

Разработка функциональной схемы является основным этапом процесса создания архитектуры ЭВМ. Здесь устанавливаются соотношения между различными уровнями организации системы в соответствии с необходимыми функциями и предъявляемыми к системе требованиями, распределяются функции между аппаратными средствами и программным обеспечением. На этом этапе принимаются решения относительно того, должна ли машина иметь стековую организацию, следует ли предусмотреть в ЭВМ операции над числами с плавающей точкой, требуется ли обработка прерываний с учетом приоритета.

На этапе разработки структурной схемы системы определяются до мельчайших подробностей «детали» ее архитектуры,

включая определение типов и форматов команд, семантики языка, способы представления данных, а также способы адресации. К сожалению, достаточно простые способы оценки эффективности принимаемых решений отсутствуют. Нельзя, например, руководствоваться такими критериями, как: «Сколько будет стоить включение команды X в набор команд?», «Что лучше: 6 или 8 регистров?», «Во что обойдется введение способа косвенной адресации?». Определенную помощь в подобных оценках может оказать статистика выполнения программ, содержащая, например, такую информацию, как частота использования конструкций языка программирования.

За разработкой структурной схемы системы обычно следует этап отладки и оптимизации последней. (Этому вопросу посвящена гл. 23.) Затем архитектура разработанной ЭВМ оценивается по ряду критериев и требований. Вопросы оценки архитектуры обсуждаются в следующем разделе данной главы и во многих последующих главах.

Отметим, что процесс проектирования архитектуры ЭВМ имеет итеративный характер, и обычно приходится многократно повторять отдельные или все этапы проектирования.

АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ЭВМ

При разработке архитектуры ЭВМ наибольшее внимание обычно уделяется проблеме ее производительности. К сожалению, эту характеристику часто неправильно понимают, и ее чрезвычайно сложно предсказать и измерить.

Основная причина неправильной интерпретации упомянутой характеристики заключается в том, что ее можно определять как на макро-, так и на микроуровне. Однако многие разработчики концентрируют свое внимание только на микроуровне. На это указывает использование таких оценок производительности, как «время выполнения операций сложения или умножения», «число миллионов операций, выполняемых в секунду». Использование оценки «число операций в секунду» имеет смысл только в том случае, когда сравниваются две реализации одной и той же архитектуры. При сравнении же подобных архитектур такой подход может легко ввести в заблуждение, а при сравнении существенно отличающихся архитектур он просто бессмыслен. В качестве примера первой ситуации можно использовать факт подобия архитектуры Системы 370 и процессора 8085 фирмы Intel (о подобии говорить возможно, если провести сравнение этих вычислительных систем с рассматриваемыми в последующих главах). Если оценивать производительность числом операций в секунду, то окажется, что микропроцессор 8085 эквивалентен по быстродействию центральному процессору ЭВМ

модели 145 Системы 370, хотя в действительности быстродействие микропроцессора 8085 значительно ниже.

Термин «производительность» трактуется неверно, поскольку его часто отождествляют со скоростью выполнения программы. Более правильным является приравнивание его к эффективности решения задач. Например, какая система имеет более высокую производительность: система, выполняющая программу X за 1 мин, но требующая 10 человеко-недель для программирования и создания отлаженной программы, или система, выполняющая ту же программу X за 2 мин при затратах на программирование 2 человеко-недель? С позиций микроуровня быстродействие первой системы по сравнению со второй в два раза выше, с позиций макроуровня — в пять раз ниже. Если же исходить из широкоизвестных статистических данных о том, что 85% всех программ выполняется только один раз (т. е. типична ситуация однократного использования программы после разработки и отладки), то такая характеристика архитектуры, как простота программирования, является более важной, чем скорость выполнения команд.

Изложенное выше не означает, что скорость выполнения программы является несущественным фактором. Важно понимать, что быстродействие зависит как от технологической базы, на которой реализуется ЭВМ, так и от архитектуры машины: первая влияет на скорость обработки данных, вторая — на объем выполняемых работ. Согласно теории вычислений, в предельном случае возможно построение ЭВМ только с двумя командами: увеличения слова на 1 и уменьшения слова на 1 и перехода, если значение слова равно 0 [5]. Нетрудно заметить, что общий объем работ (включая выборку, декодирование и выполнение команд), выполняемых такой машиной при реализации операции умножения, намного превышает объем работ машины, имеющей в наборе команд команду умножения.

Оценку эффективности архитектуры по скорости выполнения программы (например, требуется сравнить архитектуру гипотетической системы А с архитектурой существующей системы Б) непосредственно сделать нельзя по следующим причинам:

1) невозможно выполнить тестовую программу для оценки производительности, поскольку архитектура А существует только в нашем воображении;

2) можно построить имитационную модель ЭВМ с архитектурой А (используя, например, специальную программу-имитатор), но, во-первых, не ясно, какие данные, подлежащие обработке, следует использовать при моделировании; во-вторых, отсутствует информация об аппаратной реализации и стоимости производства ЭВМ с архитектурой типа А; в-третьих, всегда существует неопределенность относительно реализации модели

(например, при моделировании на ЭВМ можно предположить, что операция умножения 32-разрядных двоичных чисел со знаком выполняется за $0,5 \cdot 10^{-9}$ с, однако аппаратные средства, которыми мы располагаем, не могут обеспечить такого быстрого действия);

3) можно создать машинную реализацию архитектуры А. Однако при этом мы столкнемся с проблемой правомочности сопоставления машинных реализаций архитектур А и Б по таким параметрам, как количество электронных схем, их быстродействие и стоимость. Даже если кому-то удастся решить эту задачу или найти способ унификации средств реализации обеих машин, то в конечном счете все сводится к измерению характеристик как архитектур, так и их машинных реализаций. Кроме того, всегда остается неясным вопрос, являются ли обе машинные реализации оптимальными по отношению к своим архитектурам, и если нет, то в равной ли степени они неоптимальны;

4) описанные выше способы оценки архитектуры являются весьма дорогостоящими, требуют больших затрат времени, особенно если необходимо проанализировать большое число различных архитектурных решений.

Необходим простой и независимый от машинной реализации метод сравнения ЭВМ различной архитектуры по критерию эффективности обработки информации. Существуют простые приемы получения подобных оценок в первом приближении. Речь идет о нескольких характеристиках (параметрах), которые использовались при проведении исследования с целью нахождения простой стандартной архитектуры ЭВМ для всех тактических военных систем США [6, 7].

Хотя архитектура всех анализируемых вычислительных систем была реализована в конкретных ЭВМ, в процессе исследования изучалась только относительная эффективность самой архитектуры с целью исключения влияния на оценку эффективности определенной машинной реализации.

Анализ проводился посредством измерения следующих трех параметров:

S — размер программы;

M — количество битов, передаваемых между процессором и памятью машины (пересылки между регистрами) за время выполнения программы;

R — количество битов, передаваемых между внутренними регистрами процессора за время выполнения программы.

Параметр S — количественная оценка размера программы (определяемого длиной команд, размером косвенных адресов, объемом рабочих областей для временного размещения данных). Рациональность использования параметра S мотивирова-

лась ссылкой на него как на «важный показатель пригодности ЭВМ данной архитектуры для применения (оцениваемой по тестовой программе), который определяет объем памяти, необходимый для решения задачи» [7]. Например, архитектура микропроцессора 8080 мало подходит для копирования в памяти машинных блоков данных. Параметр S указывает на это, поскольку для выполнения операции пересылки в данном случае требуется последовательность из 4 или 5 команд, размещенных в памяти, вместо одной команды пересылки. Эта архитектура



Рис. 1.2. Модель для определения «полосы пропускания» процессора.

также неудобна для умножения чисел — отсутствует команда умножения. Для программы, в которой требуется умножение, параметр S оказывается высоким, поскольку алгоритм перемножения чисел реализуется программно.

Параметр M определяет общее количество битов информации, пересылаемых между процессором и устройством (устройствами), на котором реализуется основная память. Этот параметр является определенной оценкой предела эффективности выполнения программ почти для всех вычислительных систем, обусловленного конечностью «полосы пропускания» (bandwidth) интерфейса (произведения числа параллельных линий интерфейса на скорость передачи данных) между процессором и памятью. Таким образом, объем информации, который должен быть передан через этот интерфейс (биты команд программы и данных, пересылаемые в процессор; биты данных, пересылаемые в память), достаточно точно характеризует эффективность ЭВМ данной архитектуры. Используя модель, изображенную на рис. 1.2, можно заключить, что если выполнение некоторой программы машиной, имеющей архитектуру A , требует пересылки $6 \cdot 10^6$ бит информации, а машиной, имеющей архитектуру B , — $2 \cdot 10^6$ бит информации, то в первом приближении архитектура B обеспечивает в три раза более высокую эффективность, чем архитектура A .

Как указывалось ранее, параметр M только приблизительно характеризует ожидаемую производительность архитектуры независимо от ее машинной реализации. Важную роль играет также близость местоположения используемых данных, т. е. *локальность ссылок*. Например, эффективность работы системы существенно зависит от того, являются ли смежными по местоположению две группы битов информации, выбираемые после-

довательно. Поскольку неизвестен способ количественной оценки локальности ссылок и в связи с тем, что этот показатель зависит от машинной реализации архитектуры (разрядности интерфейса доступа к памяти, наличия кэш-памяти), в дальнейшем он рассматриваться не будет.

Параметр R используется для оценки той характеристики процесса обработки данных, которая не учитывается параметром M , а именно объема данных, передаваемых внутри процессора. Очевидно, что параметр R в значительной степени зависит от машинной реализации процессора, а также таких факторов, как организация процессора и набор функций, выполняемых арифметическо-логическим устройством (АЛУ). При проведении исследований военным ведомством для оценки параметра R была определена некая стандартная внутренняя организация процессора. В этой книге при сравнении ЭВМ различной архитектуры используются только параметры S и M . Параметр R не используется по следующим причинам:

1. Параметр R в значительной степени зависит от машинной реализации архитектуры. Поскольку рассматриваемые здесь варианты построения архитектуры ЭВМ существенно отличаются от наиболее распространенного в настоящее время решения, использование данного параметра для оценки производительности указанного разнообразия архитектурных решений не имеет смысла.

2. Некоторые весьма произвольные заключения о том, что необходимо учитывать и чего не следует учитывать при вычислении параметра R , были сделаны при проведении исследований военным ведомством [7].

3. При проведении тех же исследований было показано, что параметр M является более важной характеристикой.

4. При сравнительном анализе вычислительных систем различной архитектуры теми же исследователями установлена высокая степень корреляции между значениями параметров S , M и R (см. табл. 1.1).

Параметры S и M определяются следующим образом:

S — объем памяти в битах, занимаемый программой и обрабатываемыми ею данными;

M — количество битов информации, перемещаемых между процессором и памятью в течение времени выполнения программы.

При измерении параметра M необходимо принимать во внимание следующие обстоятельства:

1. Если архитектура ЭВМ предполагает использование адресуемых регистров, то они должны рассматриваться как часть процессора. При наличии стека необходимо в каждом случае определить, находится ли он в памяти или в процессоре.

Таблица 1.1. Измерение параметров S, M и R для 12 тестовых программ

Архитектура	S	M	R
PDP-11	1,00	0,93	0,94
Система 370	1,21	1,27	1,29
Interdata 8/32	0,83	0,85	0,83

2. При определении объема передаваемой информации всегда существуют определенные неясности, связанные с машинной реализацией архитектуры. Например, если архитектура предполагает использование числовых данных фиксированной длины в дополнительном коде и такой

команды, как «Переход, если число отрицательное», то теоретически процессору необходимо извлечь из памяти только 1 бит команды — знак. В таких случаях оценки должны делаться, исходя из наиболее рациональной и практически выполнимой машинной реализации архитектуры.

3. Существуют аппаратные средства, позволяющие уменьшить объем информации, пересылаемой между процессором и памятью, такие, как буферы и кэш-память. Они не будут рассматриваться, чтобы при сравнении ЭВМ различной архитектуры сохранить, насколько это возможно, независимость архитектурного решения от его машинной реализации.

При использовании параметров S и M желательно располагать некоторой идеальной архитектурой, с которой можно было бы сравнивать любую другую архитектуру. Эта проблема «созрела» для своего решения. Однако попытки определить идеальную архитектуру (например, [8]) сталкиваются с известными трудностями, заключающимися в том, что всегда можно предложить пути совершенствования «идеальной архитектуры». Представляется возможным определить нижнюю границу S и M следующим образом: $S=0$; M=количество битов входных данных и результатов обработки.

Другими словами, можно построить машину, для которой не требуется программы (например, единственной функцией машины является вычисление преобразования Фурье или выдача платежной ведомости служащих определенной компании). Такая машина обрабатывает входные данные только один раз, и областей памяти для временного хранения текущей информации не требуется. Для более универсальной машины, вычисляющей преобразование Фурье или выдающей платежную ведомость, нижняя граница параметра S равна 1.

Как упоминалось выше, разработчика архитектуры ЭВМ проблема производительности должна беспокоить значительно больше, и он не может ограничиться оценкой средней скорости выполнения машинных команд. Глобальный подход к определению производительности требует рассмотрения общей эффективности решения задач на ЭВМ. Даже при рассмотрении

эффективности на микроуровне необходимо уделять определенное внимание трем факторам: языку программирования, компилятору и самой машине. Быстродействие системы обеспечивает не скорость выполнения команд, а скорее «мощность» набора команд (количество функций, реализуемых машиной). Количество битов информации, передаваемых между запоминающей средой машины и процессором при выполнении данной программы, оказывают более существенное влияние на потенциальную производительность.

Считается, что разработчик архитектуры ЭВМ первоначально должен руководствоваться изложенными выше соображениями. Учитывая, что границы архитектуры ЭВМ проходят на стыке аппаратного и программного обеспечения, он должен быть хорошо знаком с этими обеими областями. Предложение [9] о том, чтобы архитектор новой машины располагал специально спроектированным, закодированным и отлаженным компилятором и операционной системой ранее разработанной машины, является недостижимым идеалом, однако такой ход рассуждений представляется вполне правомочным.

УПРАЖНЕНИЯ

1.1. Целесообразно ли определять архитектуру центрального процессора прежде, чем будет определена общая архитектура вычислительной системы?

1.2. Кто из приводимого ниже списка кандидатов является наиболее подходящим для разработки новой архитектуры ЭВМ: прикладной программист, разработчик компиляторов, разработчик операционной системы или инженер, проектирующий центральный процессор?

1.3. Проанализируйте архитектуру какой-либо известной вам ЭВМ. Имеются ли какие-нибудь свидетельства того, что при разработке этой архитектуры большое внимание было уделено достижению компромисса между функциями, реализуемыми аппаратными средствами и программным обеспечением, или система проектировалась по принципу снизу — вверх (т. е. от аппаратных средств к программному обеспечению)?

1.4. Приводимые далее две программы, составленные на языке ассемблера, обрабатывают 16-разрядные числа, расположенные последовательно в памяти, до тех пор, пока не встретится число, равное 0. Слева приведена программа для ЭВМ Системы 370, справа — для ЭВМ 8080 фирмы Intel.

	LA	R3,1		LXI	D,0001
	L	R1,START		LHLD	START
LOOP	LH	R2,0(R1)		MVI	A,0
	LTR	R2,R2	LOOP	MOV	B,M
	BZ	FOUND		INX	H
	LA	R3,1(R3)		CMP	B
	LA	R1,2(R1)		JNZ	END
	B	LOOP		MOV	B,M
				CMP	B
			END	JZ	FOUND
				INX	D
				INX	H
				JMP	LOOP

Предполагая, что нулю равно четвертое число (первыми четырьмя числами являются 0001, 0F9D, 0005 и 0000), определите параметры S и M для обеих программ.

1.5. Предположите, что приведенные выше программы являются типичными и позволяют судить обо всех других программах (что, вообще говоря, является неверным предположением). Что вы можете сказать: 1) об относительном объеме памяти, необходимом для архитектуры каждой из систем; 2) о скорости выполнения программ при эквивалентной машинной реализации обеих архитектур?

1.6. Какая часть численного значения параметра M относится к обработке данных в случае программы Системы 370?

1.7. Укажите несколько причин, обуславливающих различие численных значений параметров S и M для двух рассматриваемых архитектур.

1.8. Позволяют ли параметры S и M приведенных программ адекватно сопоставлять архитектуру вычислительных систем 8080 и 370?

1.9. Какие из ранее определенных критериев оценки архитектуры (стоимость, быстродействие, надежность, защита от несанкционированного доступа) вам менее всего понятны?

ЛИТЕРАТУРА

1. Myers G. J., *Digital System Design with LSI Bit-Slice Logic*, New York, Wiley, 1980.
2. Reddi S. S., Feustel E. A., *A Conceptual Framework for Computer Architecture*, *Computing Surveys*, 8(2), 277—300 (1976).
3. IBM System/370 Principles of Operation, GA22-7000, IBM Corp., Pooughkeepsie, NY, 1974.
4. Thurber K. J., *Techniques for Requirements-Oriented Design*, *Proceedings of the 1977 NCC*, Montvale, NJ, AFIPS Press, 1977, pp. 919—929.
5. Minsky M. L., *Computation: Finite and Infinite Machines*, Englewood Cliffs, NJ, Prentice-Hall, 1967.
6. Dietz W. B., Szeferenko L., *Architectural Efficiency Measures: An Overview of Three Studies*, *Computer*, 12(4), 26—33 (1979).
7. Burr W. E. et al., *Computer Family Architecture Via Test Programs*, ECOM-4528, Army Electronics Command, Fort Monmouth, NJ, 1977.
8. Flynn M. J., *A Canonic Interpretive Program Form for Measuring 'Ideal' HLL Architectures*, *Computer Architecture News*, 6(8), 6—15 (1978).
9. McKeeman W. M., *Language Directed Computer Design*, *Proceedings of the 1967 Fall Joint Computer Conference*, Washington, Thompson, 1967, pp. 413—417.

ГЛАВА 2

КРИТИКА ТРАДИЦИОННОЙ АРХИТЕКТУРЫ ФОН НЕЙМАНА

Главной предпосылкой для написания данной книги явилось несоответствие архитектуры большинства современных вычислительных систем тому ее определению, которое дано в гл. 1. Вместо того чтобы конструировать систему, исходя из целостности ее принципов и неразрывности взаимодействия аппаратных средств — программного обеспечения, большинство разработчиков ЭВМ используют традиционный подход и проектируют систему «снизу вверх», стремясь минимизировать стоимость аппаратных средств и возлагая на плечи программистов решение всех остальных трудных проблем.

Подтверждением сказанного является отсутствие (за небольшими исключениями) в архитектуре современных вычислительных машин существенных изменений и усовершенствований по сравнению с архитектурой ЭВМ 50-х годов. Такое утверждение, звучащее как обвинение, может, конечно, вызвать немедленное возражение с указанием на появление и внедрение принципов микропрограммирования, поточной (конвейерной) обработки команд, кэш-памяти, больших и сверхбольших интегральных схем. Однако все перечисленные усовершенствования не являются примерами различных принципов организации архитектуры вычислительной системы. Скорее это достижения в реализации уже имеющихся архитектурных решений — в организации процессора или вычислительного процесса. Более того, некоторые из этих достижений можно рассматривать как шаг назад в процессе поиска наилучшей архитектуры вычислительной системы. Например, внедрение принципа поточной обработки повлекло за собой появление так называемого «неопределенного» прерывания [1]. Дело в том, что ЭВМ поточной обработки оперирует параллельно группой последовательных команд и даже может обрабатывать независимые команды, не принадлежащие данной последовательности. Как следствие этого, при возникновении прерывания ЭВМ иногда не может

однозначно определить, какая именно команда вызвала прерывание, и не может гарантировать невыполнение команды, непосредственно следующей за той, которая вызвала прерывание. При работе с ЭВМ поточной обработки программисты получают не вполне достоверную информацию о допущенных ошибках и предоставлены самим себе в решении вопроса относительно того, что же неверно.

Если сравнить архитектуру большинства наиболее широко используемых современных ЭВМ (например, Система 34 и Система 370 фирмы IBM, ЭВМ PDP-11, VAX-11, Univac 1108, машина 8085 фирмы Intel) с архитектурой первых электронных машин с запоминаемой программой EDVAC и EDSAC (построенных в 40-х годах), то оказывается, что появление всех существенных различий датируется 50-ми годами. Эти отличительные особенности вычислительных машин, появившихся после машины EDVAC, сводятся к следующим:

1. *Индексные регистры.* Позволяют формировать адреса памяти добавлением содержимого указательного регистра к содержимому поля команды. Этот принцип впервые реализован в 1949 г. в ЭВМ Манчестерского университета и использован в 1953 г. фирмой Electro Data Corporation при производстве ЭВМ Datatron.

2. *Регистры общего назначения.* Благодаря этой группе регистров устраняется различие между индексными регистрами и аккумуляторами и в распоряжении пользователя оказывается не один, а несколько регистров-аккумуляторов. Впервые это решение нашло свое воплощение, по-видимому, в ЭВМ Pegasus фирмы Ferranti (1956 г.).

3. *Представление данных в форме с плавающей точкой.* Представление данных в виде мантиссы и порядка и выполнение операций над ними было реализовано в 1954 г. в вычислительных машинах NORC и 704 фирмы IBM.

4. *Косвенная адресация.* Средство, позволяющее использовать команды, указывающие адреса, по которым в свою очередь находится информация о местоположении операндов команд. Принцип косвенной адресации был реализован в 1958 г. в ЭВМ 709 фирмы IBM.

5. *Программные прерывания.* При возникновении некоторого внешнего события состояние вычислительной системы, связанное с выполнением прерванной команды, запоминается в определенной области. Этот принцип впервые был применен в 1954 г. в машине Univac 1103.

6. *Асинхронный ввод-вывод.* Параллельно обычному выполнению команд независимые процессоры управляют операциями ввода-вывода. Первой ЭВМ с независимым процессором ввода-вывода являлась ЭВМ 709 фирмы IBM (1958 г.)

7. *Виртуальная память.* Определение адресного пространства программы осуществляется без «привязки» к физическим областям памяти обычно с целью создания впечатления, что вычислительная система имеет больший объем основной памяти, чем тот, которым она фактически располагает. В 1959 г. в вычислительной системе Atlas Манчестерского университета были реализованы принципы разделения памяти на страницы и динамическая трансляция адресов аппаратными средствами.

8. *Мультипроцессорная обработка.* Два или более независимых процессора обрабатывают потоки команд из общей памяти. Не вполне ясно, кому принадлежит приоритет введения этого способа обработки информации, однако в конце 50-х и начале 60-х годов он был реализован в вычислительных машинах Sage фирмы IBM, Sperri-Univac LARC и D825 фирмы Burroughs.

Хотя существующие в настоящее время вычислительные системы значительно отличаются от своих предшественниц стоимостью, быстродействием, надежностью, внутренней организацией и схемотехническими решениями, архитектура большинства из них не претерпела изменений с 50-х годов. В основном она определяется традициями и устаревшими представлениями о практической реализации. Эти традиции восходят к первым ЭВМ с запоминаемой программой 40-х годов, когда основным требованием было решение задач численного анализа (первой подобной задачей было решение нелинейных дифференциальных уравнений) посредством простых программных средств общего назначения. Вполне понятно, что возможности разработчиков ЭВМ были ограничены требованиями минимизации стоимости и сложности архитектурных решений. При этом программное обеспечение и связанные с ним проблемы рассматривались не иначе как курьез.

Изложенное выше, естественно, должно вызвать следующие вопросы:

1. Являются ли оптимальными на все времена архитектурные решения, предложенные в 40 — 50-х годах?

2. Достаточные ли изменения претерпели соответствующие технологические возможности современного мира (стоимость и быстродействие логических схем, диапазон областей применения вычислительных средств, важность проблем программного обеспечения), чтобы считать оправданным внесение существенных изменений в архитектуру ЭВМ?

СЕМАНТИЧЕСКИЙ РАЗРЫВ

Поскольку второй важной предпосылкой для написания данной книги явилось наличие серьезных недостатков в архитектуре современных ЭВМ, прежде чем давать рекомендации по их

устранению, целесообразно проанализировать эти недостатки. Большинство из них подпадает под определение феномена, известного как *семантический разрыв* и предложенного к использованию в качестве меры различия принципов, лежащих в основе языков программирования высокого уровня, и тех принципов, которые определяют архитектуру ЭВМ [2]. Как правило, современные вычислительные системы имеют нежелательно большой семантический разрыв, выражающийся в том, что объекты манипулирования и соответствующие им операции, реализуемые архитектурой вычислительной системы, редко имеют близкое «родство» с объектами и операциями, описываемыми в языках программирования. Расширим это определение утверждением, что существует большой разрыв в семантике между «программным окружением» ЭВМ и представлением принципов построения программ на уровне ее архитектуры или, развивая эту мысль далее, между архитектурой машины и средой, в которую ее помещают для использования. Как будет показано ниже, этот большой семантический разрыв обуславливает возникновение большого количества существенных проблем, к которым относятся высокая стоимость разработки программного обеспечения, его ненадежность и эксплуатационная неэффективность, чрезмерный объем программ, сложность компиляторов и операционной системы, наличие отступлений от правил построения языков программирования, т. е. факторы, отрицательно влияющие на экономический показатель вычислений.

Чтобы понять наличие семантического разрыва (согласно его определению по отношению к языку программирования и архитектуре машины), можно выбрать какой-либо язык и архитектуру и заняться изучением взаимосвязей между ними. Для этих целей подходит почти любой используемый в настоящее время язык программирования и архитектура любой машины. В качестве примера проанализируем язык ПЛ/1 и Систему 370. Выводы, сделанные в этом случае, будут носить более общий характер, поскольку большинство принципов, положенных в основу ПЛ/1, справедливы и для таких языков, как КОБОЛ, ФОРТРАН и Паскаль, а Система 370 отображает большинство традиционных архитектурных решений и, кроме того, знакома широкой аудитории читателей.

Указанный анализ предлагается проводить следующим образом: определить принципы, положенные в основу языка программирования, и попытаться измерить «расстояние» между ними и соответствующими принципами, на которых базируется анализируемая архитектура вычислительной системы. Ниже перечисляется несколько основных и широко используемых принципов построения языков программирования, в том числе рас-

смагнваемого языка ПЛ/1. Ставится задача определения адекватных принципов, заложенных в архитектуру Системы 370.

Массивы. Это наиболее часто используемый тип организации данных в языке ПЛ/1 и большинстве других языков. Язык ПЛ/1 предоставляет такие возможности, как использование многомерных массивов, обращение к отдельным элементам массива посредством индексов, операции над целыми массивами, обращение к отдельным подмассивам внутри массива, защиту от выхода за пределы соответствующего массива. В языке Ада эти принципы организации массивов расширены дополнительными возможностями, в частности возможностью соединения массивов.

Анализ архитектуры Системы 370 показывает, что в ней не предусмотрены средства, соответствующие описанным принципам организации массивов. Исключением можно считать принцип использования индексных регистров, являющийся примитивным подобием одной из возможностей языка. Следовательно, реализация широко используемого принципа организации данных в виде массивов возлагается на компилятор, в распоряжении которого имеется набор команд Системы 370, весьма далекий по своим возможностям от задач, возникающих при работе с массивами. (Более подробно эта проблема рассматривается в конце данной главы, в упр. 2.1.)

Структуры. Это второй тип часто используемой организации данных в виде наборов разнородных элементов данных (известных в некоторых языках программирования под названием *записей*). И в данном случае в Системе 370 отсутствует средство, адекватное структурам и операциям над ними.

Обработка строк. В языке ПЛ/1 применяются так называемые *строковые данные* (или просто *строки*) фиксированной и переменной длины. Над строками допускается выполнение таких операций, как слияние, выделение заданной части (подстроки) из исходной строки, поиск строки по заданной подстроке, определение текущей длины строки, проверка присутствия элементов одной строки в другой строке. Подобных возможностей в архитектуре Системы 370 не предусмотрено. Конечно, поскольку существуют компиляторы ПЛ/1 для упомянутой системы, указанные возможности этого языка реализуются посредством компиляторов, однако при использовании весьма примитивных команд Системы 370. При использовании Системы 360 проблемы, «стоящие перед компиляторами», еще сложнее, поскольку команды этой вычислительной системы могут оперировать содержимым областей памяти объемом не более 256 байт, в то время как строки в ПЛ/1 могут достигать длины 65 534 байт. Задача манипулирования строками битов (возможность, предоставляемая языком ПЛ/1) при реализации в Систе-

ме 370 осложняется тем, что последняя допускает адресацию только к группам из 8 бит (т. е. к байтам).

Процедуры. Основной структурной единицей языка ПЛ/1 является процедура или подпрограмма. При вызове процедуры требуется сохранение состояния текущей процедуры, динамическое назначение памяти для локальных переменных вызванной процедуры, передача параметров и инициализация выполнения вызванной процедуры. В Системе 370, по существу, отсутствуют возможности, соответствующие этим принципам организации программ на ПЛ/1. Незначительным исключением является команда BALR (ПЕРЕХОД С ВОЗВРАТОМ), но ее вклад в реализацию операции вызова процедуры столь незначительный (это одна из многих команд, подлежащих выполнению при вызове), что ее отсутствие могло бы остаться незамеченным. Компилятор мог бы заменить ее двумя другими командами: LA (ЗАГРУЗКА АДРЕСА) и BR (ПЕРЕХОД БЕЗУСЛОВНЫЙ).

Блочная структура. Язык ПЛ/1 имеет блочную структуру, предполагающую существование правил определения границ областей действия идентификаторов, присваиваемых объектам. Эти правила определяют возможность адресации необъявленных (не получивших имена) переменных ссылок во внутренних блоках. Ничего подобного этому принципу адресации в Системе 370 нет, и, следовательно, реализация этого принципа должна осуществляться генерированием компилятором соответствующего машинного кода.

ON-блоки¹⁾. В языке ПЛ/1 предусмотрена возможность обработки программных прерываний, вызываемых так называемыми особыми ситуациями: можно указывать для определенных исключительных ситуаций обрабатывающие их ON-блоки; динамически назначать и отменять действия ON-блоков; определять область их действия в блоках и вызываемых процедурах. В Системе 370 близким указанным возможностям языка ПЛ/1 можно считать механизм контроля программных прерываний. Однако в этой вычислительной системе прерывания обрабатываются безотносительно к конкретной программе или процессу, т. е. в реализации ON-блоков должна участвовать операционная система. Кроме того, многие особые ситуации в ПЛ/1 (например, CONVERSION, SUBSCRIPTRANGE, ERROR, NAME, CHECK, SIZE) не имеют соответствующих прерываний в Системе 370, в связи с чем задача генерирования машинного кода обнаружения и обработки этих ситуаций возлагается на компилятор.

¹ ON-блок — это условное обозначение действий, которые программист может организовать на языке ПЛ/1 с помощью оператора ON при возникновении особой ситуации, вызывающей программное прерывание — *Прим. ред.*

Представление данных. В языке ПЛ/1 используется представление десятичных и двоичных данных в форме с фиксированной точкой (целая часть . дробная часть). В Системе 370 подобной формы представления нет: десятичные и двоичные данные имеют вид целых величин, а на компилятор возлагается обязанность реализации представления данных в форме с фиксированной точкой. При записи на языке ПЛ/1 десятичные числа могут содержать от 1 до 15 цифр. Что же касается Системы 370, то здесь допускается представление десятичных чисел, содержащих только нечетное количество цифр. Двоичные числа на языке ПЛ/1 могут состоять из любого количества цифр в пределах от 1 до 31; в Системе 370 количество цифр двоичного числа не должно выходить за диапазон значений 15—31. При записи на языке ПЛ/1 чисел в форме с плавающей точкой количество значащих цифр числа должно находиться в пределах 1—33; однако представление этих чисел в Системе 370 требует использования одной из трех форм записи фиксированной длины.

Подобные рассуждения можно было бы продолжать бесконечно, анализируя такие принципы, заложенные в основу языка ПЛ/1, как управляемая память (стековая организация памяти), вызов общих процедур, функции отслеживания программы и автоматическое преобразование данных. Однако приведенных здесь примеров, очевидно, достаточно для понимания того, что такое семантический разрыв между принципами языков программирования высокого уровня и принципами архитектуры современных ЭВМ.

Значение рассмотренных средств программирования зависит от того, насколько широко они используются. Даже если и существует большой подобный разрыв между принципами языка X и архитектурой соответствующей ЭВМ, при крайней ограниченности использования подобных средств языка и машины значимость выявленного семантического разрыва крайне мала.

Информация, характеризующая программы, весьма ограничена, но даже те данные, которые имеются, указывают на сравнительно частое применение средств программирования, упомянутых в приведенных выше примерах. Так, согласно статистике, массивы относятся к типичной структуре данных. В соответствии с работой [3] 9,2% всех обращений к операндам фактически является обращением к элементам массивов. В другом исследовании [4] указывается, что 45% всех арифметических операторов имеют дело по крайней мере с одним массивом или элементом массива. Даже согласно анализу программ на языке КОБОЛ [7], в процессе выполнения программы 30% всех обращений к операндам являются обращениями к элементам массива.

Обращения к процедурам (подпрограммам) используются чаще, чем это принято считать. Авторы работы [3] утверждают, что 16% выполняемых операторов языка высокого уровня — это обращения к подпрограммам-процедурам, подпрограммам-функциям (указателям функций) и операторы возврата. В той же работе отмечено, что стандартная процедура (подпрограмма) состоит из восьми или девяти операторов описания, четырех обращений к другим процедурам, одного цикла и одного оператора окончания (например, оператора возврата в вызвавшую процедуру). Исследование работы компилятора языка Bliss показывает, что такой компилятор затрачивает 25% своего рабочего времени на установление связей между подпрограммами [5]; для компилятора языка ФОРТРАН подобные затраты времени равны 15%. По данным другого исследования [7] операторы CALL, RETURN и PROCEDURE составляют 19% всех операторов программы.

Статистические данные о программах свидетельствуют о том, что в программировании ярко выражена тенденция решения «нечисленных» задач. В соответствии с работой [4], 18% всех используемых переменных представляют собой строки символов, а 24% всех операторов присваивания предназначены для символьной информации. Обширные исследования программ на языке ПЛ/1, проведенные фирмой General Motors [8], показывают, что 27% всех используемых констант — это строки символов, а не числовые данные. При изучении программ на КОБОЛе [6] обнаружено, что 48% всех данных — строки символов со средней длиной, равной 24 символам.

СЕМАНТИЧЕСКИЙ РАЗРЫВ МЕЖДУ АРХИТЕКТУРОЙ ЭВМ И ОПЕРАЦИОННОЙ СИСТЕМОЙ

Операционная система — неотъемлемая часть большинства современных вычислительных систем. В общем случае операционная система выполняет следующие четыре функции:

1) предоставляет другим программам определенный вид обслуживания (посредством программ-утилит), например выделение и назначение памяти, синхронизацию процесса вычислений и организацию взаимосвязи между различными процессами в вычислительной системе;

2) обеспечивает защиту (в определенной мере) других программ от последствий различных особых ситуаций, возникающих при машинной реализации данной программы, таких, как прерывания и машинные сбои;

3) реализует с той или иной степенью сложности принцип «виртуальной машины», что позволяет группе программ использовать общие вычислительные ресурсы, например процессор (процессоры) и основную память;

4) организует и следит за выполнением принципов управления при решении таких задач, как обеспечение защиты данных от несанкционированного доступа и реализация системы приоритетов доступа программ к вычислительным ресурсам.

Для демонстрации семантического разрыва между принципами, положенными в основу операционных систем, и принципами, использованными при построении соответствующих вычислительных машин, можно воспользоваться теми же приемами, которые применялись выше для иллюстрации разрыва между языками программирования и архитектурой соответствующих машин. Например, можно сослаться на тот факт, что многие разработчики операционных систем признают решающую роль модели так называемых *рабочих наборов* [9] для близкого к оптимальному управлению иерархией памяти (например, страничной организацией). Но хотя известны средства реализации такой модели [10], они отсутствуют в архитектуре ЭВМ, выпускаемых промышленностью.

Однако, по-видимому, центральным принципом, «порожденным» операционными системами, является понятие процесса как неделимого целого — единицы параллельных операций и «обладателя» вычислительных ресурсов. В то же время большинство современных ЭВМ построено так, как будто этот принцип не получил признания. Задачи синхронизации процессов и установления между ними взаимосвязи, решаемые посредством так называемых семафоров, критических секций и мониторов, а также механизмов приема-передачи, не находят своего воплощения в интерфейсе ЭВМ.

На операционную систему возлагается также задача по разделению информации между «пользователями» и ее защите от несанкционированного доступа. Хотя многие архитектурные решения ЭВМ потенциально ориентированы на решение подобной задачи, на практике она обычно реализуется без достаточной гибкости и относительно произвольно. Например, Система 370 обеспечивает защиту памяти поблочно — 2048 байт памяти образуют один блок. Это средство используется для защиты программ друг от друга, но оно бесполезно, если необходимо защитить и обеспечить раздельное использование подпрограмм и переменных.

СЕМАНТИЧЕСКИЙ РАЗРЫВ МЕЖДУ АРХИТЕКТУРОЙ ЭВМ И ПРИНЦИПАМИ ПОСТРОЕНИЯ ПРОГРАММНЫХ СРЕДСТВ

Признаки большого семантического разрыва можно также обнаружить и между принципами, определяющими архитектуру ЭВМ, и принципами построения ее программных средств. Например, архитектура современных ЭВМ не содержит никаких

специальных аппаратных средств, ориентированных на реализацию таких важных принципов, закладываемых при проектировании в программное обеспечение больших систем, как модульность, сокрытие информации, использование абстрактных типов данных и мониторов. Хорошо известно, что 50% и более денежных средств, затрачиваемых на разработку больших программ, расходуется на проверку и отладку последних, в то время как архитектура современных ЭВМ предоставляет весьма мало средств для решения этой проблемы.

Хотя разработчики ЭВМ принимают во внимание возможность отказов аппаратных средств, создается впечатление, что они считают программы свободными от ошибок. Если и предпринимались какие-то меры для исправления неудовлетворительного положения в указанной области (например, микропроцессор 68000 фирмы Motorola содержит команду BOUNDS-CHECK, используемую для проверки значения индекса элемента массива относительно верхней и нижней границ последнего), то они столь незначительны, что программист, имеющий дело с программой, содержащей ошибки, если и получает от аппаратных средств используемой ЭВМ какую-либо помощь, то она крайне мала.

СЕМАНТИЧЕСКИЙ РАЗРЫВ МЕЖДУ АРХИТЕКТУРОЙ ЭВМ И ОРГАНИЗАЦИЕЙ ПАМЯТИ

Этот разрыв обнаружить значительно труднее, чем другие семантические разрывы, поскольку он представляется несуществующим. И не потому, что проектировщики ЭВМ уже «навели мост через эту пропасть», а по той причине, что, в сущности, разработчики языков и операционных систем «свалились» в нее.

До сих пор отсутствует единообразный подход к организации памяти. Архитектура современных ЭВМ такова, что программист, как правило, имеет несистематизированное, хаотичное представление о памяти машины. Память воспринимается им как некая кажущаяся иерархия, в соответствии с которой организуются регистры или стеки, запоминающее устройство произвольного доступа (основная память), диски, барабаны и ленты. Запоминающая среда каждого типа имеет свою систему адресации и распределения памяти, свой механизм защиты и набор функциональных возможностей. Так, применительно к Системе 370 для обращения к данным в основной памяти программа должна содержать линейное смещение, добавляемое к адресу, хранимому в специальном регистре. Если же данные размещены где-либо в другом месте (например, на дорожке магнитного диска), то необходимо применить совершенно другой принцип адресации (требуется использовать номера диско-

вода, цилиндра, дорожки, записи, а также линейное смещение в пределах записи). Для выполнения над некоторыми данными такой операции, как сложение, программисту необходимо принимать во внимание ту запоминающую среду, в которой находятся данные, ибо операция сложения определена только для подмножества всего разнообразия запоминающих сред. Если данные размещены в основной памяти, то сложение можно выполнить непосредственно; если же они находятся на диске, то требуется не только прибегнуть к другому типу адресации, но и создать специальную программу (цепочки управляющих слов канала), чтобы при использовании Системы 370 скопировать данные в основную память, что необходимо для выполнения операции сложения.

Вместо того чтобы сформулировать новые принципы решения подобной проблемы, создатели языков программирования разработали модели организации памяти, повторяющие те, которые заложены в архитектуру соответствующих ЭВМ. В результате семантического разрыва не существует, но его отсутствие нельзя признать положительным решением проблемы. Программисту, работающему на языках высокого уровня, предоставляется не единая систематизированная структура организации памяти, а некоторое количество невязанных принципов организации, определяемых возможностями технологии производства и практической применимостью.

Отметим, что принципы виртуальной памяти в том виде, в каком они воплощены в современных вычислительных системах, не решают указанной здесь проблемы. Чаще всего виртуальная память создает у программиста лишь иллюзию наличия большего объема основной памяти, чем она есть в действительности, а это ничего не добавляет к решению проблемы отсутствия единого подхода к организации памяти.

ПОСЛЕДСТВИЯ СЕМАНТИЧЕСКОГО РАЗРЫВА

Простое указание на наличие семантического разрыва нескольких типов не обязательно влечет за собой необходимость внесения улучшений в архитектуру ЭВМ. Потребность в них становится очевидной в результате анализа последствий, к которым приводит указанный семантический разрыв. Этому вопросу и посвящены следующие ниже разделы.

НЕНАДЕЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Семантический разрыв существенно влияет на ненадежность программного обеспечения в том смысле, что значительная доля ошибок программирования, которые теоретически могли бы

быть предотвращены или обнаружены вычислительной системой, не выявляются системами, находящимися в эксплуатации в настоящее время. Ограничимся здесь лишь несколькими примерами. Более детальное обсуждение этого предмета будет сделано в последующих главах, особенно в части V.

Типичной ошибкой программирования, возникающей в самых разнообразных ситуациях, является обращение к переменной, значение которой не определено или не установлено. Большинство современных вычислительных систем не обнаруживает подобной ошибки, а поскольку выполнение программы продолжается и используется непредсказуемое значение переменной (программа начинает носить недетерминированный характер), выявить и устранить ошибку чрезвычайно трудно. Причина последнего обстоятельства заключается в том, что, хотя на этапе компилирования в какие-то моменты ошибка и может быть обнаружена путем анализа хода алгоритма, в общем случае выявление ошибки возможно не ранее наступления этапа выполнения. Поскольку традиционные машины не располагают средствами установления факта присутствия (или отсутствия) значения переменной, эта проблема оказывается переложеной с разработчика архитектуры ЭВМ на разработчика компилятора. А так как последний не находит простого и эффективного решения этой проблемы, он в свою очередь перекладывает ее решение на программиста — составителя прикладных программ, что делает описываемую ошибку наиболее часто встречающейся в современном программировании [11].

При создании некоторых компиляторов предпринимались попытки решения указанной проблемы, однако используемые средства оказались сложными, неэффективными и не предоставляли возможности подтверждения отсутствия ошибок. Например, компилятор языка ПЛ/1 с диагностикой ошибок, разработанный фирмой IBM, присваивает в качестве начального значения всем строкам символов значение шестнадцатеричных символов FE (знак *m* спецификации формата), всем двоичным числам с фиксированной точкой — значение наименьшего отрицательного числа, а затем проверяет эти значения при каждом обращении к указанным переменным. Однако введение такой меры не только увеличивает расход машинного времени и памяти, но может привести к обнаружению «ошибок» в правильных программах; при этом описываемый метод не охватывает все типы переменных. Позже будет показано, что рассматриваемая ошибка программирования может быть выявлена буквально без всяких затрат при соответствующем выборе архитектуры ЭВМ.

Другой типичной ошибкой является обращение к элементу массива, один из индексов которого выходит за соответствующую

щие границы. Поскольку для традиционной машины такого понятия, как массив, не существует, и в этом случае также ответственность за решение данной проблемы возлагается на разработчика компилятора. Так как простые ответы на вопрос, поставленный последнему, не известны, эта проблема либо игнорируется вовсе, либо ее решение путем введения соответствующего контроля возлагается на программиста, составляющего прикладные программы. Но для установления такого контроля требуются существенный дополнительный объем памяти и машинное время. Поэтому в руководстве по программированию разработчик компилятора предупреждает: «С целью экономии памяти и машинного времени рекомендуется использовать подобный контроль только на этапе тестового прогона программ с последующим удалением средств контроля из конечного продукта — рабочей программы» [12]. Такое решение может вызвать возражения, поскольку оно «подобно предложению моряку пользоваться спасательным жилетом при тренировках на суше, а при выходе в море не надевать его» [13]. Другими словами, кажется странным предложение о проверке наличия ошибок на этапе тестовых прогонов программы (когда ошибочные результаты вовсе не опасны) и удалении средств проверки из рабочей программы (когда ошибочный результат может привести к катастрофическим последствиям).

Язык Ада тоже предоставляет программисту возможность изымать некоторые или все средства контроля из программы в процессе ее выполнения. Однако к достоинствам этого языка следует отнести, во-первых, функционирование контроля, если последний специально не отменен, и, во-вторых, отсутствие требования к компилятору просматривать программу с целью поиска запросов на подавление средств контроля.

Примером затрат машинного времени и памяти на выполнение контроля программными средствами является работа компилятора языка ПЛ/1 фирмы IBM, генерирующего 17 машинных команд (занимающих 62 байт памяти) для оператора

$$C(I,J)=A(I,J)+(B,I);$$

где A, B и C — массивы двоичных элементов одинакового размера в форме с фиксированной точкой. Если необязательное средство контроля SUBSCRIPTRANGE подключено, то компилятор генерирует 75 машинных команд (274 байт); причем из них выполнению подлежит лишь 57 команд, если значения индексов переменных не выходят за допустимые пределы. Следовательно, это средство контроля увеличивает объем памяти, занимаемый объектным кодом указанного оператора, до 340%, а время выполнения оператора возрастает в три раза. В дальнейшем будет показано, что подобный контроль может быть вы-

полнен машиной без всякого увеличения объема памяти и затрат машинного времени. (Машина, обнаруживающая такую ошибку, вероятно, должна быть более быстродействующей, как показано в гл. 4, поскольку в ее архитектуре отражен принцип организации данных в форме массивов.)

Дж. Вейзенбаум в книге, посвященной этике вычислений [14], при описании работы ЭВМ в терминах, понятных специалистам, указывает на существование семантического разрыва и его взаимосвязь с надежностью:

«Работа вычислительных машин поразительно легко нарушается чисто техническими, т. е. лингвистическими, программными ошибками, однако эти нарушения таковы, что их истинная причина скрыта, замаскирована. Причиной затруднений, с которыми сталкиваются при программировании, является в большинстве случаев то обстоятельство, что машине в действительности ничего не известно о тех сторонах реального мира, с которыми программа должна иметь дело... (так, недопустимо высказывание типа 2,999 человек). Довольно трудно объяснить что-либо (например, лингвистические концепции) в терминах примитивного словарного запаса (т. е. машинными командами), не имеющего ничего общего с тем, что подлежит объяснению... Для написания хорошего сонета или хорошей программы необходимо понимать, что нужно выразить. И конечно, благоприятна ситуация, когда критик (машина) располагает знаниями, адекватными тем, которые находятся в распоряжении объекта критики (языка программирования)».

ПРОБЛЕМЫ ЭФФЕКТИВНОСТИ ЭВМ

Большой семантический разрыв порождает также и значительные проблемы, связанные с уменьшением эффективности работы вычислительной машины вследствие того, что, располагая только довольно примитивным набором машинных команд для реализации возможностей языка программирования, компилятор вынужден генерировать, а машина интерпретировать большое число команд. Это снижает скорость выполнения задания на ЭВМ, поскольку увеличивается объем информации, которой обмениваются память и процессор. Как показано в гл. 1, величина этого объема является хорошим критерием первого порядка для сравнения производительности различных машин.

Рассмотрим отмеченное явление на простом примере. Допустим, что необходимо сложить две матрицы целых чисел размером 100×100 . Средствами языка ПЛ/1 решение можно сформулировать в следующем виде: $A = A + B$; использование для этих целей вложенного цикла DO значительно более неэффективно. Компилятор языка ПЛ/1 Системы 370 генерирует для

этого оператора эффективный объектный код — шесть команд, за которыми следует цикл из четырех команд, выполняемых 10 000 раз. Общее количество 32-битовых слов, подлежащих перемещению между памятью и процессором, равно 70 007. Из этого количества 40 004 — это команды, первые шесть из которых укладываются в четыре слова, и четыре слова занимает тело цикла, а 30 003 — это данные. Значения двух данных извлекаются из памяти, а результат операции над ними записывается в память при обработке каждой пары элементов массивов (к этому добавляется еще несколько выбираемых из памяти данных). Однако если бы компилирование выполняла машина, в архитектуре которой отражен принцип организации данных в форме массивов (например, машина, описываемая в части V), то могло бы оказаться достаточно одной машинной команды (прибавление В к А) при объеме слов, перемещаемых между памятью и процессором, порядка 30 000 (к этой величине следует добавить одно слово для команды и, возможно, еще несколько для информации, описывающей массивы). Кроме того, если первая из упомянутых машин должна декодировать и интерпретировать 40 004 команды, то вторая только декодирует одну команду. Конечно, имеются и другие показатели эффективности работы машины, кроме количества пересылок между памятью и процессором и объема работ по декодированию команд (например, вычисление адресов и выполнение арифметического сложения), но эти показатели примерно одинаковы для обеих машин.

Хотя рассмотренный пример справедлив только для операций с массивами, можно найти аналогичные примеры генерирования чрезвычайно большого количества команд при реализации на традиционных ЭВМ почти каждого принципа, положенного в основу языка программирования.

Семантический разрыв между языком программирования и архитектурой машины при описании и реализации процедур ввода-вывода также вносит определенный вклад в общую неэффективность вычислительных систем. Вместо того чтобы снабдить программиста памятью, единой по своим принципам организации на всех уровнях иерархии, современные средства вычислительной техники понуждают программиста управлять памятью с учетом уровня иерархии. Иначе говоря, в настоящее время иерархией памяти системы управляют прикладные программы, вследствие чего достигается не глобальная, а лишь локальная оптимизация использования вычислительных ресурсов. Примером может служить двойное буферирование — технический прием, обеспечивающий эффективное решение задачи оптимизации для одиночной программы, но вовсе не обязательно оптимально использующий вычислительные ресурсы системы

при одновременном выполнении нескольких программ. Зачем растрчивать циклы выполнения команд, которые управляют буферами или перемещают записи, если вместо этого можно было бы использовать машинное время на выполнение другого процесса?

ЧРЕЗМЕРНО БОЛЬШОЙ РАЗМЕР ПРОГРАММЫ

Рассматриваемый семантический разрыв отрицательно сказывается не только на эффективности работы машины, но и на размере программы: компиляторы должны генерировать большое количество машинных команд, чтобы компенсировать последствия этого разрыва. Например, ранее упоминалось, что для представления оператора

$$C(I,J)=A(I,J)+B(I,J);$$

необходимо 62 байт памяти, если не требуется контроля значений индексов, и 274 байт, если подобный контроль желателен. При использовании машины, описываемой в части V, этот оператор может быть представлен двумя машинными командами, занимающими вместе 13 байт памяти (при автоматическом выполнении указанного контроля).

Несмотря на постоянные заявления о том, что «дефицит памяти вскоре исчезнет», потребность в ее увеличении все еще остается одной из проблем, с которой сталкиваются при использовании ЭВМ. Это объясняется несколькими причинами. Во-первых, стоимость памяти относительно высока. Так, стоимость основной памяти микропроцессорной системы часто превосходит стоимость процессора на порядок и более. Во-вторых, потребность в памяти увеличивается примерно с той же скоростью, с которой падает ее стоимость. В-третьих, чрезвычайно большой размер программ, обусловленный неэффективностью организации вычислительной системы в целом, вызывает необходимость в рабочих наборах¹.

СЛОЖНОСТЬ КОМПИЛЯТОРА

Материал, рассмотренный в двух предыдущих подразделах, делает очевидным наличие семантического разрыва между принципами, лежащими в основе архитектуры машины, и принципами, определяющими построение компилятора языка программирования. Механизм генерирования машинных кодов в компиляторе должен быть чрезвычайно сложным, чтобы эффективно ликвидировать семантический разрыв.

Может возникнуть вопрос о том, может ли сокращение раз-

¹ По-видимому, речь идет о наборах страниц виртуальной памяти, одновременно находящихся в реальной памяти. — *Прим. ред.*

рыва устранить указаниую сложность структуры компилятора, поскольку эта сложность где-то локализуется. Если разрыв сокращается за счет усложнения архитектуры машины, то упомянутая сложность «переместится» из компилятора в машину. Подобный обмен упрощает структуру компилятора по двум причинам. Во-первых, если допустить, что вычислительная система располагает более чем одним компилятором (для Системы 370 написаны сотни компиляторов), то семантический разрыв должен быть преодолен либо в каждом компиляторе, либо однократно при реализации машины. Например, если всем используемым системой языкам программирования присущи такие средства, как описания массивов и вызовы процедур, то представляется более разумным (даже без учета других возможных сопутствующих преимуществ) ликвидировать этот разрыв один раз путем соответствующей модификации архитектуры машины, чем делать это для каждого компилятора. Во-вторых, желаемого результата можно добиться, если решение этой проблемы частично возложить на машину, а частично — на компилятор с обеспечением хорошего интерфейса (согласования) между этими частями. В этом случае решение, как правило, достигается более простыми средствами.

ИСКАЖЕНИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ И ЕГО НЕКОРРЕКТНОЕ ИСПОЛЬЗОВАНИЕ

Если семантический разрыв столь велик, что компилятор не может эффективно преодолеть его полностью, то в определении языка программирования появляются искажения (нарушения правил его построения), которые в машине, реализующей язык, учесть нельзя. Вследствие этого возможно некорректное использование языка.

В качестве примера предположим, что при работе с Системой 370 с использованием языка ПЛ/1 программист объявляет переменную *A* десятичным числом с фиксированной точкой, состоящим из двух цифр (разрядов). При использовании оператора присваивания $A=100$ естественно было бы ожидать сообщения об ошибке. Однако этого не происходит, и при выводе значения *A* на печать появится число 100. Объясняется это тем, что Система 370 может представлять десятичные числа, имеющие только нечетное количество цифр. Предвидя, что полное устранение семантического разрыва привело бы к генерированию неэффективных объектных кодов, разработчики компилятора предпочли преобразование двухразрядных десятичных переменных языка ПЛ/1 в трехразрядные операнды машины. Другая возможная альтернатива — изменение соответствующего правила языка ПЛ/1 — могла бы в равной степени приводить

в замешательство программистов и повлечь за собой излишнюю ориентацию языка на архитектуру Системы 370.

Для иллюстрации рассматриваемой проблемы можно обратить внимание на тот факт, что язык ПЛ/1 допускает десятичное и двоичное представления чисел. Трудно понять, зачем программисту, работающему на этом языке, пользоваться двоичным представлением, поскольку если в его распоряжении находится переменная M , определенная как двоичное число с фиксированной точкой, то для присвоения ей значения 25 необходимо воспользоваться оператором присваивания

$$M=11001B;$$

Большинство лиц, программирующих на языке ПЛ/1, по невнимательности записывают $M=25$, но 25 — число десятичное, поэтому средства языка должны запрашивать автоматическое преобразование данных, использование которого при недостаточном опыте может повлечь за собой много странных ошибок [15]. Однако при работе с Системой 370 на языке ПЛ/1 многие программисты пользуются двоичными числами потому, что десятичные числа занимают больший объем памяти, а выполнение десятичных операций требует значительно большего времени. Следовательно, архитектура используемой ЭВМ побуждает программиста к некорректному использованию языка программирования.

В качестве еще одного примера применительно к языку ПЛ/1 рассмотрим использование переменных в формате строки битов. Неискушенный программист, пытающийся передать значение такой переменной в другую процедуру через список параметров, часто сталкивается с неожиданным результатом. Поскольку в Системе 370 допустима адресация только на границу байта, а строка битов вовсе не обязательно начинается с такой границы, между двумя процедурами возможно возникновение проблемы выравнивания данных на границу памяти. Читатель, имеющий опыт программирования на языке ПЛ/1, возможно, знает, что атрибут `ALIGNED` был добавлен к средствам этого языка для решения проблем подобного типа. Это — еще одно указание на влияние архитектуры машины на язык программирования.

Искажения можно найти и в других языках. Например, в языке ФОРТРАН арифметический оператор `IF` передает управление одному из трех операторов в зависимости от значения вычисляемого выражения — является ли оно отрицательным, равным нулю или положительным. Это связано не с эстетическими концепциями структуры языка программирования, а с наличием в ЭВМ 704 фирмы IBM команды сравнения, по результату выполнения которой управление передается одной из

трех последующих команд. Другой особенностью языка ФОРТРАН является тот факт, что для цикла, формируемого посредством оператора DO, всегда необходимо выполнение по крайней мере одной итерации. Это требование также обусловлено машиной, реализующей Фортран-программу, а именно использованием в ЭВМ 704 команды TIX.

Язык Ада — свидетельство компромиссов, которые необходимы для организации надежного программирования при использовании современных машин. Последние не обеспечивают корректное использование типов переменных (допуская, например, применение в командах несовместимых операндов, не реагируя на подмену формальных параметров фактическими и наоборот) и не выявляют обращения за допустимое адресное пространство (не обнаруживают, что используемый адрес более недействителен). Для устранения этих ошибок и ошибок других типов разработчики языка Ада предпочли ограничить гибкость языка так, чтобы подобные ситуации можно было предотвратить или обнаружить на этапе компилирования. В результате подобных компромиссов язык становится менее гибким (язык ПЛ/1 содержит некоторые средства программирования, отсутствующие в языке Ада), но более надежным (меньшее количество ошибок остается необнаруженным) при использовании машин современной архитектуры.

НИЗКАЯ ПРОДУКТИВНОСТЬ ПРОГРАММИРОВАНИЯ

Стоимость разработки программного обеспечения стала главным препятствием на пути дальнейшего роста объема вычислений. В то время как достигнуто значительное снижение стоимости аппаратных средств вычислительной техники, затраты на разработку программного обеспечения остаются относительно постоянными. Частично это объясняется наличием семантического разрыва.

Примером существенного препятствия повышению продуктивности программирования является семантический разрыв, обусловленный отсутствием общего, единого подхода к организации памяти ЭВМ. Большинство трудностей, которые возникают при использовании прикладных программ, связаны с управлением памятью, т. е. этим программам должно быть известно о том, что основной файл записан на диске, и их команды должны явным образом осуществлять пересылку записей между диском и основной памятью. Можно с уверенностью сказать, что программист, разрабатывающий прикладную программу, затрачивает больше времени на управление памятью и перемещение данных, чем на преобразование последних. Согласно упомянутому ранее анализу программ на языке ПЛ/1 [8], каждый

из 20 операторов — оператор ввода или вывода. Для программ на языке КОБОЛ [6] 19% всех операторов программы являются операторами ввода или вывода. Если бы можно было отказать от процедур ввода-вывода, стоимость разработки программ сократилась бы вдвое и более.

Анализ затрат на разработку программного обеспечения содержится в последующих главах. Как отмечает Дж. Деннис [16], «нельзя ожидать значительных улучшений методологии разработки очень больших программ при отсутствии существенных изменений в структуре аппаратных средств и организации вычислительных систем».

ОГРАНИЧЕНИЯ НА ВЫБОР ВАРИАНТА ПОСТРОЕНИЯ МАШИНЫ

Еще одним следствием наличия семантического разрыва является появление определенных ограничений на организацию процессора. Применение принципа параллелизма операций представляется важнейшим способом преодоления пределов быстрой работы аппаратных средств, однако архитектура низкого уровня ограничивает возможности проектировщика в его стремлении использовать данный принцип только двумя вариантами обработки — *мультипроцессорной* и *поточной*. Располагая архитектурой низкого уровня с такими примитивными командами, как LOAD REGISTER (ЗАГРУЗКА РЕГИСТРА) и ADD INTEGER (СЛОЖЕНИЕ ЦЕЛОЧИСЛЕННОЕ), конструктор не способен найти применение принципа параллелизма, кроме как частично перекрывая во времени выполнение последовательных команд. Другое решение — мультипроцессорная обработка — приводит лишь к частичному успеху из-за проблем, возникающих при ее реализации (в частности, перекрытия областей памяти и сложности синхронизации), и проблем программирования (трудность декомпозиции задачи на параллельно решаемые подзадачи и сложность реализации в компиляторах средств обнаружения в программах последовательностей команд, подлежащих параллельному выполнению) [17].

В последующих главах показано, что сокращение семантического разрыва требует от машины большего «понимания» того, что она делает: ее операции должны одновременно охватывать значительно больший фрагмент общей работы. Последнее предоставляет инженеру широкие возможности для использования принципа параллелизма.

Подводя итоги, можно сказать, что свойственный современным вычислительным системам семантический разрыв порождает много существенных недостатков, которые в свою очередь отрицательно сказываются на экономических факторах разработки и использовании вычислительных систем.

АРХИТЕКТУРА ФОН НЕЙМАНА

Основная причина существования большого семантического разрыва в современных вычислительных системах заключается в том, что их архитектура, по существу, не отличается от архитектуры модели ЭВМ, предложенной Дж. фон Нейманом в 40-х годах [18]. Это не означает, что архитектуре фон Неймана не были присущи признаки гениального творения. Но со времени ее создания мир претерпел громадные изменения. Поскольку в то время практическая пригодность даже конструируемых электронных вычислительных машин ставилась под сомнение и главное внимание обращалось на стоимость аппаратных средств и надежность машины, основная задача заключалась в проектировании процессора настолько примитивным, насколько это возможно. А такие столь существенные для сегодняшнего дня факторы, как наличие языков программирования высокого уровня, сложность и «чувствительность» (к побочным явлениям) подлежащих решению задач, не только не принимались в то время во внимание, но и оставались незамеченными. И если учесть, что архитектура фон Неймана разрабатывалась для решения определенной проблемы — обеспечения пользователя простым устройством с запоминаемой программой, предназначенным для выполнения вычислений с целью решения дифференциальных уравнений, — остается только удивляться, что эта архитектура сохранилась в наши дни.

По существу, все современные ЭВМ можно классифицировать как машины фон Неймана. Принято считать, что машине с архитектурой фон Неймана присущи следующие характеристики:

1. Единственная последовательно адресуемая память. Программа и данные хранятся в одной памяти, адреса областей которой составляют последовательность типа 0, 1, 2, ...
2. Память является линейной. Она одномерная, т. е. имеет вид вектора слов.
3. Отсутствует явное различие между командами и данными. Их идентифицируют неявным способом при выполнении операций. Так, объект, адресуемый командой перехода, определяется как команда; операнды, с которыми имеет дело команда сложения, определяются как данные. Эти принимаемые по умолчанию соглашения позволяют, например, обращаться с командой как с данными (в частности, модифицировать ее), складывать команду со словом данных или осуществлять переход к слову данных и выполнять его так, как будто бы биты этого слова представляют команду.
4. Назначение данных не является их неотъемлемой составной частью. Нет, например, никаких средств, позволяющих явно

отличить набор битов, представляющих число с плавающей точкой, от набора битов, являющихся строкой символов. Назначение данных определяется логикой программы. Если машина извлекает из памяти команду сложения чисел с плавающей точкой, то предполагается, что операнды — числа с плавающей точкой, и над операндами выполняется сложение согласно правилам арифметики чисел с плавающей точкой. Следовательно, можно выполнить подобное сложение над двумя операндами, являющимися в действительности, например, строкой символов и адресом.

Хотя архитектура фон Неймана была логичным решением проблемы создания первой машины с запоминаемой программой, она не удовлетворяет требованиям, которые выдвигает задача выполнения программ, написанных на языках высокого уровня. В отличие от перечисленных выше четырех характеристик языки высокого уровня имеют следующие характеристики:

1. Память, согласно ее представлению в языке высокого уровня, состоит из набора дискретных именуемых переменных. За исключением некоторых вызывающих разногласие конструкций языка (таких, как общая область в языке ФОРТРАН), здесь отсутствует принцип размещения одной переменной рядом с другой. Нет никаких оснований полагать, что переменные одной подпрограммы расположены в том же запоминающем устройстве, что и переменные другой подпрограммы. Таким образом, принцип единственной последовательной памяти мало напоминает принцип организации памяти, формулируемый в соответствии с требованиями языков программирования.

2. Языки программирования оперируют многомерными, а не просто линейными данными (в языках имеются массивы, структуры, списки).

3. Языкам программирования присуще резкое разграничение между данными и командами. В большинстве языков отсутствует возможность обработки данных или обращения к командам, как будто бы они данные.

4. В языках высокого уровня назначение данных является внутренней частью самих данных. Вместо записи вида

DECLARE A WORD;

DECLARE B WORD;

A=A «сложение с плавающей точкой» B;

в программах записывают

DECLARE A DECIMAL FLOAT (6);

DECLARE B DECIMAL FLOAT (6);

A=A+B;

Иначе говоря, в языках высокого уровня назначение данных связано с самими данными; данные определяют и опера-

ции, выполняемые над ними (например, действие символа «+» определяется свойствами операндов этой операции).

Итак, принципы, на которых основывается архитектура фон Неймана, не согласуются с принципами языков программирования и даже им противоречат. Машина фон Неймана оказывается плохим средством для выполнения программ, написанных на языках высокого уровня, по следующим причинам:

1. Чрезмерный расход программных средств (например, работа компилятора по формированию машинных кодов) с целью согласования возможностей языка со структурой памяти по фон Нейману, что трактуется как «абсорбирование структуры (данных) в логике программы» [19]. Это становится очевидным каждому, анализирующему результат работы компилятора: объем машинных кодов, генерируемых компилятором с целью отражения языковой концепции памяти и данных в архитектуре используемой машины, обычно значительно превосходит объем подобных кодов, предназначенных непосредственно для решения поставленной задачи.

2. Машина фон Неймана — чрезмерно универсальна. Так, можно использовать слово, значение которого для текущего момента не определено, адресоваться к чему угодно в памяти, складывать строку символов с командой. А поскольку подобная универсальность отсутствует в языках программирования, на компилятор (и генерируемый им код) возлагается задача устранения универсальности и обеспечения отсутствия искажений, которую она может внести в определение языка.

3. В силу относительной примитивности принципа организации памяти по фон Нейману операции (набор команд), выполняемые машиной, оказываются в равной мере примитивными.

Подобно другим первенцам науки и техники, таким, как электромеханические реле и электровакуумные лампы в логических схемах, логарифмическая линейка, теория строения атома Н. Бора, перфоратор, архитектура ЭВМ фон Неймана сослужила добрую службу человечеству и к настоящему времени изжила себя. Требования к вычислительным системам достигли высокого уровня сложности, причем их большая часть сосредоточена на комплексе программных средств. Стремление к удовлетворению постоянно растущих требований заставляет «архитектора» вычислительных систем совершенствовать взаимосвязь архитектуры машины с программным обеспечением, т. е. улучшать архитектуру машины таким образом, чтобы она лучше согласовывалась с программными средствами.

Прежде чем завершить обсуждение архитектуры фон Неймана, отметим, что структура современных языков программирования имитирует основные принципы организации машины фон Неймана в виде арифметического устройства, устройства

управления и памяти [20]. Так, переменная имитирует пассивное запоминающее устройство, оператор присваивания — арифметическое устройство, последовательное выполнение операторов управления (таких, как GO TO, IF) — устройство управления. Более подробно об этом речь идет в гл. 22.

ДОПОЛНИТЕЛЬНЫЕ ПРИЧИНЫ СЕМАНТИЧЕСКОГО РАЗРЫВА

Хотя основной вклад в наличие семантического разрыва вносит сама модель машины фон Неймана, дополнительным источником причин этого разрыва являются некоторые характеристики архитектуры современных вычислительных систем.

ДВОИЧНАЯ АРИФМЕТИКА

В современных машинах место двоичной арифметики, можно сказать, священо, несмотря на то что человек воспринимает ее почти во всех случаях с определенным чувством неприязни. Можно спорить относительно того, является ли это следствием определенной культуры, присуще ли это природе человека или порождено математикой. Но факт неприятия человеком двоичной арифметики налицо. Поскольку предложение перехода к десятичной арифметике часто вызывает горячие споры, бесполезно проанализировать все «за» и «против».

Два довода можно привести в пользу десятичной арифметики. Первый из них основан на том обстоятельстве, что в настоящее время применение ЭВМ сопряжено с большим объемом работ по организации ввода-вывода. А поскольку найдется немного сторонников выполнения этих работ с использованием представления чисел в двоичной системе счисления, то современные вычислительные системы вынуждены затрачивать огромное количество времени на десятично-двоичные преобразования. (Предвидя это, фон Нейман справедливо полагал, что пользователи ЭВМ изъявят желание воспользоваться восьмью и шестнадцатеричными системами счисления [18].)

Второй из упомянутых выше доводов в пользу десятичной системы счисления базируется на следующем. От пользователя ЭВМ не удастся полностью скрыть тот факт, что числа внутри машины представлены в двоичной форме, поскольку, например, дробные части вещественных чисел представляются как бесконечные дроби двоичной системы счисления. Это означает, что двоичные числа конечной длины — часто лишь аппроксимации десятичных чисел и, как следствие, причина затруднений при программировании, источник ошибок в программе и некорректности некоторых формальных определений синтаксиса языка

программирования. Вот, например, к каким последствиям это приводило при использовании бортовых ЭВМ при полетах американских космических кораблей по программе «Аполлон». При вводе данных в машину посредством клавиатуры ЭВМ должна воспроизводить вводимые данные на экране дисплея в целях визуального контроля. В руководстве оператора для подобного случая имелось следующее предупреждение: «Не следует беспокоиться, если воспроизводимые на экране данные отличаются от вводимых с клавиатуры цифрой младшего разряда, поскольку ЭВМ выполняет вычисления посредством двоичной арифметики».

Несмотря на то что при создании языка Ада обращалось особое внимание на проблемы точности вычислений и портативности программы (простоты переноса программы из одной вычислительной среды в другую), операции этого языка над данными вещественного типа (числами с фиксированной или плавающей точкой) выполнимы только при наличии в машине двоичной арифметики, а следовательно, заведомо предполагается аппроксимация числовых данных (хотя и при наличии средств управления ошибкой аппроксимации). По традиции программисты проявляют определенную осторожность при использовании данных в форме с плавающей точкой, и все же многих шокирует то обстоятельство, что арифметика с плавающей точкой неизбежно сопряжена с аппроксимацией числовых данных. В то же время, например, в языках ПЛ/1 и КОБОЛ для десятичных чисел арифметика с фиксированной точкой выполняется точно.

При реализации в языке Ада определения

```
type UNIT__PRICE is delta 0.01 range 0.00...500.00;
```

переменные этого типа могут быть представлены как 16-разрядные целые двоичные числа, причем каждое приращение (интервал) равно 0.0078125. Описания и оператор присваивания

```
PAPER__COST:UNIT__PRICE:=0.30;  
TOTAL__COST:UNIT__PRICE;  
TOTAL__COST:=PAPER__COST.10;
```

присваивают переменной TOTAL__COST значение 2.97 или 3.05 (в зависимости от той или иной реализации языка); потери 3% или 5% являются неприемлемыми.

Традиционными аргументами против десятичной арифметики являются следующие: она уступает в быстродействии двоичной арифметике; двоичные числа можно хранить в более компактной форме, чем десятичные. Но эти доводы подлежат обсуждению. Во-первых, при оценке скорости выполнения ариф-

метических операций следует учитывать необходимость предварительного преобразования десятичных чисел в двоичные и последующего обратного преобразования. Во-вторых, разработаны схемы выполнения операций десятичной арифметики (например, [21]), быстродействие которых может конкурировать с быстродействием схем двоичной арифметики; первые незначительно уступают последним только по показателю стоимости. Новые быстродействующие большие интегральные схемы (БИС) на основе логических схем со связанными эмиттерами, предназначенные для выполнения арифметических операций, способны оперировать двоично-кодированной десятичной информацией с той же скоростью, что и БИС двоичной арифметики [22].

Второй аргумент против десятичной арифметики (неэкономное использование пространства памяти) более существенный, но и его нельзя считать непреодолимым. Одним из путей решения этой проблемы можно считать применение представления чисел, отличного от двоично-кодированного десятичного. Если вместо основания 10 использовать основание 100, то для хранения двух десятичных цифр окажется достаточным иметь семь двоичных разрядов вместо восьми [23]. Переход к основанию 1000 позволяет размещать три десятичные цифры в 10 двоичных разрядах вместо 12. Подобное представление десятичных чисел для арифметики с плавающей точкой предлагалось ранее [24].

РЕГИСТРЫ

Использование программно-адресуемых регистров, получивших название *регистров общего назначения*, является еще одной концепцией архитектуры современных вычислительных систем, чуждой принципам языков программирования. Обычно эти регистры определяются следующими характеристиками:

- 1) предназначены для формирования самостоятельного адресного пространства, отличного от адресного пространства основной памяти ЭВМ;
- 2) ограниченным количеством (например, в машине может быть предусмотрено 8 или 16 регистров);
- 3) фиксированным количеством двоичных разрядов регистра (16 или 32 разряда);
- 4) адресуется посредством коротких адресов;
- 5) адресуется за более короткое время, чем подобные элементы основной памяти;
- 6) управляются программными средствами при выполнении всех процедур и вычислительных процессов, являясь тем самым единственным подобным ресурсом системы;

7) функционально специализированы до определенной степени. (Это означает, что при реализации некоторых конкретных функций требуется использование регистров, иногда вполне определенных.)

Принцип использования программно-адресуемых регистров плохо согласуется со структурой организации данных в современных языках программирования и влечет за собой следующие проблемы:

1. *Слишком частое использование команд LOAD (ЗАГРУЗКА В РЕГИСТР) и STORE (ЗАПИСЬ В ПАМЯТЬ).* Во многих вычислительных системах, использующих регистры общего назначения (Система 370, PDP-11), арифметические операции и операции адресации могут быть выполнены только посредством этих регистров. Поток команд подобных вычислительных систем изобилует командами LOAD и STORE, вследствие чего возрастает размер программ и снижается скорость их выполнения. Об этом речь пойдет далее в этом разделе, а также в гл. 4.

2. *Неэффективность процедур сохранения и перезагрузки содержимого регистров при обращениях к подпрограммам.* Набор регистров — единственный ресурс вычислительной системы, содержимое которого частично или полностью подлежит сохранению при вызове подпрограмм и обработке прерываний с последующей перезагрузкой этого содержимого при возврате к вызывавшему программному модулю или модулю, выполнение которого прерывалось. В дополнение к этому можно заметить, что управление регистрами посредством программных средств приводит к выполнению многих ненужных операций сохранения и перезагрузки их содержимого. Часто стратегия управления такова, что та или иная подпрограмма предполагает сохранение и восстановление содержимого тех конкретных регистров, которые ею используются. Поскольку подпрограмма «не знает», как эти регистры используются программными модулями, ее вызывающими, возможно выполнение сохранения и восстановления содержимого регистров, которыми указанные программные модули не пользуются. Нерациональность сохранения и перезагрузки содержимого регистров проявляется и при повторении последовательности обращений к подпрограммам, например к трем подпрограммам А, В и С. Когда выполняется возврат из подпрограммы А, содержимое регистров восстанавливается только для того, чтобы вновь быть загруженным подпрограммой В на сохранение, и т. д. Подобные непроизводительные операции имеют место и при работе с набором вложенных подпрограмм: подпрограмма D возвращает управление подпрограмме С, которая тотчас же возвращает управление подпрограмме В, последняя сразу же возвращает управление под-

программе А и т. д. При возвращении управления каждая подпрограмма восстанавливает содержимое регистров, и при этом большая часть этой работы оказывается совершенно ненужной.

3. *Потеря общности принципов программирования.* Имеются в виду некоторые проблемы программирования и организации работы компилятора, возникающие вследствие различия механизма адресации регистров и памяти. В качестве примера рассмотрим подпрограмму, получающую некоторый параметр при адресации к ней. Если данная подпрограмма вызывается другой подпрограммой, использующей в качестве этого параметра локальную переменную, то необходимо либо отказаться от хранения локальной переменной в регистре (что может быть нежелательным при частом ее использовании), либо хранить эту переменную в регистре, помещать ее на временное хранение в память перед вызовом подпрограммы, передавать адрес области временного хранения и загружать ее обратно в регистр после возврата из подпрограммы.

4. *Проблемы программирования.* Они обусловлены тем, что широко используемые регистры общего назначения представляют собой память, структура которой отлична от структуры основной памяти вычислительной системы. Если последняя обычно имеет побайтовую адресацию, то к регистрам адресуются как к словам (регистр — это запоминающее устройство объемом, как правило, 16 или 32 бит). Все благополучно до тех пор, пока объем данных совпадает с емкостью регистров. Последние бесполезны или почти бесполезны, когда необходимо манипулировать битами, байтами, цепочками символов и т. п. Регистры — неотъемлемый ресурс программистов и разработчиков компиляторов, а поэтому и те и другие должны владеть техникой манипуляции этим ресурсом. Ошибки при работе с регистрами часто ведут к «загадочным» ошибкам. Поскольку механизм функционирования регистров носит специальный характер, у программиста возникают дополнительные трудности. Например, при работе с микропроцессором Motorola 68000 индексирование можно выполнить только посредством адресных регистров. Более того, многие операции этой машины осуществимы только при участии регистров данных, а не адресных регистров. В Системе 370 некоторые операции определены только для операндов, размещаемых в регистрах, другие — для операндов как хранимых в памяти, так и загруженных в регистры, и, наконец, имеются операции, выполняемые только над данными в основной памяти.

5. *Проблемы отладки.* Регистры часто используются для размещения локальных и временных переменных. Если программа прервана или каким-либо иным способом остановлена

в процессе выполнения, программист не всегда способен идентифицировать точное состояние программы.

6. *Сложность архитектурных решений.* Порождается спецификой средств адресации и функционирования регистров. А поскольку последние — неотъемлемая часть процессора, возникает неизбежное переплетение различных архитектурных решений в рамках единой вычислительной системы.

После этих критических замечаний небесполезно проанализировать исходные доводы в пользу регистров общего назначения. Во-первых, их введение преследовало цель сокращения размера программ. Адреса регистров обычно меньше адресов основной памяти, и, как следствие, команды, манипулирующие содержимым регистров, компактнее команд, адресуемых к памяти. Во-вторых, поскольку регистры выполняются на элементах памяти повышенного быстродействия, предполагалось достижение большей скорости выполнения операций.

Упомянутое сокращение размера программ — спорный вопрос. Во-первых, необходимо найти определенный баланс между возможностью использования более коротких команд и сопутствующей потребностью в дополнительных командах по перемещению данных из регистров в память и обратно (команды **LOAD** и **STORE**). Анализ программ, написанных на различных языках программирования для ЭВМ PDP-10, показывает, что 42% всех выполняемых команд затрачивается на перемещение данных между памятью и регистрами [5, 25]. Исследование возможностей оптимизирующих компиляторов для семейств ЭВМ PDP-11 и CDC 6000 свидетельствует о незначительном влиянии на размер программ сокращения числа регистров [26, 27]. Во-вторых, в современных машинах форма адресации памяти носит слишком общий характер и не имеет тесной взаимосвязи со средствами адресации, используемыми программами. Совершенствование механизма адресации (см. гл. 4) может привести к сокращению в командах размера адреса памяти, что в свою очередь снижает эффективность использования регистров в целях подобного сокращения.

Следует обсудить и второй довод в пользу введения регистров — достижение большей скорости выполнения команд. И в этом случае необходим определенный компромисс между получением более быстрого доступа к данным в регистрах и сопутствующим требованием заблаговременного извлечения и декодирования команд перемещения данных между регистрами и памятью. Объем работ по такому извлечению и декодированию команд может оказаться значительным (42% от общего объема команд согласно упомянутому выше исследованию). Иллюстрацией сказанного может служить и табл. 2.1, которая содержит сведения о количестве различных команд небольшой

Таблица 2.1. Некоторые статистические сведения о командах программы, выполненной Системой 370

Мнемоническое название команды	Количество выполненных команд	Количество извлеченных байтов команды	Количество извлеченных из памяти и записанных в нее байтов данных
BC	77	308	0
L	45	180	180
LA	44	176	0
ST	41	164	164
SR	24	48	0
LH	22	88	44
MVC ¹⁾	20	120	158
LR	18	36	0
LM ²⁾	15	60	368
TM	15	60	15
CR	13	26	0
BCR	13	26	0
BALR	12	24	0
ZAP	10	60	99
LTR	8	16	0
OI	7	28	14
STM ³⁾	6	24	280
AR	6	12	0
STH	5	20	10
NI	5	20	10
EX	4	16	0
MVZ	4	24	12
CLC	4	24	8
MVN	3	18	9
CVB	3	12	24
LD	3	12	24
STD	3	12	24
CVD	3	12	24
BXLE	3	12	24
MVI	2	8	2
CH	2	8	4
SLL	2	8	0
OR	2	4	0
SH	2	8	4
AH	2	8	4
CP	2	12	10
Команды, выполненные один раз	18	68	63
Всего	468	1770	1578

¹⁾ Команды MVC осуществляли пересылку 1—8 байт.²⁾ Команды LM производили загрузку 2—14 регистров.³⁾ Команды STM выполняли запись в память содержимого 9—11 регистров.

программы на языке ПЛ/1 для Системы 370. Команды расположены в порядке убывания частоты их использования и их вклада в значение параметра М.

Они могут быть разделены на две группы: команды, выполняющие работу согласно алгоритму решаемой задачи, и команды подготовки к подобной работе. К последним можно отнести все команды загрузки регистров и записи их содержимого в память, а также все команды перехода (единственное назначение которых — изменение содержимого счетчика команд). При рассмотрении программы с такой точки зрения оказывается, что 65% команд выполнения операций и 69% команд перемещения данных между процессором и памятью — это команды подготовки к выполнению работ, предписываемых алгоритмом исходной задачи. Подобная статистика не является нетипичной. Более обширные исследования программирования в Системе 370 [7] показывают, что удельный вес подобных команд в программе достигает 70,1%.

Для более детального знакомства с использованием регистров можно рекомендовать исследования А. Лунде [5, 25], демонстрирующие ряд программ и компиляторов, выполняемых на машине с регистрами. Одним из выводов этих исследований является утверждение, что для машины с 16-разрядными регистрами общего назначения среднее число задействованных регистров (т. е. таких, в которых что-либо содержится в данный момент или будет находиться в ближайшем будущем) составляет только 3,9. Другой вывод сводится к тому, что значительное сокращение числа регистров в машине оказывает несущественное влияние на скорость выполнения программы.

Эффект высокой скорости регистровых операций снижается стремлением современного программирования к достижению максимальной модульности комплекса программ, приводящей к дроблению каждой программы на большое количество отдельных компилируемых подпрограмм. Это обстоятельство оказывает двойное воздействие на использование регистров. Во-первых, уменьшается эффект оптимального использования регистров, на достижение которого ориентирована работа компилятора. Во-вторых, наличие большого количества программных модулей влечет за собой увеличение частоты операций загрузки регистров и записи их содержимого в память (как следствие, например, необходимости перехода от одного программного модуля к другому).

Отметим, что даже при отказе от использования регистров общего назначения, возможно достижение любого быстродействия, обеспечиваемого их применением. Так, реализация принципа кэш-памяти позволяет получить такую же высокую скорость доступа к данным, которую обеспечивают регистры, и из-

бежать при этом большинства проблем, упомянутых выше. Дальнейшему анализу архитектуры вычислительной машины, в которой используются регистры общего назначения, посвящена гл. 4.

УПРАЖНЕНИЯ

2.1. Назовите пять характеристик массивов (таблиц данных) в языке ПЛ/1 и детально поясните причины семантического разрыва между соответствующей структурой языка и организацией памяти машины фон Неймана.

2.2. Хотя в данной главе и проводится идея о необходимости сокращения семантического разрыва, на практике существуют пределы такого сокращения. Какая может возникнуть основная проблема при значительном сокращении семантического разрыва?

2.3. Несмотря на то что еще не были сформулированы архитектурные принципы сокращения семантического разрыва, перечислите четыре характеристики машины фон Неймана и применительно к каждой из них укажите соответствующие меры возможного сокращения семантического разрыва.

2.4. Почему в большинстве языков программирования на современных машинах $1,0$ редко равно $10,0 \times 0,1$?

2.5. Насколько менее эффективно кодирование десятичных цифр четырьмя битами по сравнению с использованием двоичной системы счисления?

2.6. Прокомментируйте целесообразность построения машины с наборами регистров общего назначения — один набор для каждого вычислительного процесса или уровня прерываний — при условии, что преследовалась цель сокращения действий по загрузке и запоминанию содержимого регистров при переходе от одного процесса (процедуры) к другому.

2.7. Для программы, представленной в табл. 2.1, вычислите отношение числа загрузок регистров (предположим, при использовании команд L, LH, LM, LA, LR) к числу обращений к содержимому регистров, исключая операции записи в память (например, две в команде AR или CR, одна в команде A или BCR). (Пренебрегите использованием регистров базы для переопределения местоположения программы; считайте, что в этой программе никакие операции с индексными регистрами не выполняются.)

2.8. Возьмите программу средних размеров (или несколько небольших примеров фрагментов программ) и проведите количественный анализ по использованию языковых конструкций. Важной является статистика по использованию декларативных и императивных операторов, различных типов данных, сложности выражений и среднего числа обращений к каждой переменной.

2.9. Рекомендуется прочесть оригинальную статью фон Неймана о машине с запоминаемой программой (она включена как гл. 4 в книгу Белла и Ньюэла [28]). В этой статье обсуждаются некоторые из проблем, упомянутых в настоящей главе, и объясняется, почему пришлось идти на те или иные компромиссы.

ЛИТЕРАТУРА

1. Anderson D. W., Sparacio F. J., Tomasulo R. M., The IBM System/360 Model 91: Machine Philosophy and Instruction Handling, *IBM Journal of Research and Development*, 11(1), 8—24 (1967).
2. Gagliardi U. O., Report of Workshop 4 — Software-Related Advances in Computer Hardware, Proceedings of a Symposium on the High Cost of Software, Menlo Park, CA: Stanford Research Institute, 1973, pp. 99—120.

3. Tanenbaum A. S., Implications of Structured Programming for Machine Architecture, *Communications of the ACM*, 21(3), 237—246 (1978).
4. Wortman D. B., A Study of Language Directed Computer Design, Ph. D. dissertation, Stanford University, Stanford, CA, 1972.
5. Lunde A., Empirical Evaluation of Some Features of Instruction Set Processor Architecture, *Communications of the ACM*, 20(3), 143—153 (1977).
6. Chevance R. J., Heidet T., Static Profile and Dynamic Behavior of Cobol Programs, *SIGPLAN Notices*, 13(4), 44—57 (1978).
7. Alexander W. G., Wortman D. B., Static and Dynamic Characteristics of XPL Programs, *Computer*, 8(11), 41—46 (1975).
8. Elshoff J. L., An Analysis of Some Commercial PL/I Programs, *IEEE Transactions on Software Engineering*, SE-2(2), 113—120 (1976).
9. Denning P. J., The Working Set Model for Program Behavior, *Communications of the ACM*, 11(5), 323—333 (1968).
10. Morris J. B., Demand Paging Through Utilization of Working Sets on the MANIAC II, *Communications of the ACM*, 15(10), 867—872 (1972).
11. Myers G. J., The Design of Computer Architectures to Enhance Software Reliability, Ph. D. dissertation, Polytechnic Institute of New York, 1977.
12. OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, SC33-0009, IBM Corp., White Plains, NY, 1972.
13. Knuth D. E., Structured Programming with GO TO Statements, *Computing Surveys*, 6(4), 261—301 (1974).
14. Weizenbaum J., Computer Power and Human Reason: From Judgment to Calculation, San Francisco, Freeman, 1976.
15. Myers G. J., Software Reliability: Principles and Practices, New York, Wiley, 1976.
16. Dennis J. B., Computer Architecture and the Cost of Software, *Computer Architecture News*, 5(1), 17—21 (1976).
17. Dennis J. B. et al., Research Directions in Computer Architecture, MIT/LCS/TM-114, MIT Laboratory for Computer Science, Cambridge, MA, 1978.
18. Burks A. W., Goldstine H. H., von Neumann J., Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, Pt. 1, vol. 1, Institute for Advanced Study, Princeton, NJ, 1946.
19. Illife J. K., Basic Machine Principles, 2nd ed, London, Macdonald, 1972.
20. Backus J., Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, 21(8), 613—641 (1978).
21. Schmookler M. S., Weinberger A., High Speed Decimal Addition, *IEEE Transactions on Computers*, C-20(8), 862—866 (1971).
22. Myers G. J., Digital System Design with LSI Bit-Slice Logic, New York, Wiley, 1980.
23. Chen T. C., Ho I. T., Storage-Efficient Representation of Decimal Data, *Communications of the ACM*, 18(1), 49—52 (1975).
24. Ris F. N., A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages, *SIGPLAN Notices*, 12(9), 60—70 (1977).
25. Lunde A., Evaluation of Instruction Set Processor Architectures by Program Tracing, Ph. D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1974.
26. Yuval G., Is Your Register Really Necessary? *Software-Practice and Experience*, 7(2), 295—296 (1977).
27. Yuval G., The Utility of the CDC 6000 Registers, *Software-Practice and Experience*, 7(2), 535—36 (1977).
28. Bell C. G., Newell A., Computer Structures, New York, McGraw-Hill, 1971.

ГЛАВА 3

ПРИВЯЗКА ПРОГРАММ К МАШИНАМ

Возможные пути сокращения семантического разрыва между языком программирования и машиной отличаются друг от друга тем, как и за какое время осуществляется так называемая *привязка программы к машине*, на которой программа должна выполняться. Привязку можно определить как процесс дополнения и модификации программы посредством информации, позволяющей выполнять программу на выбранной машине [1]. Привязку можно также рассматривать как выбор определенных характеристик программы из некоторого набора возможных характеристик.

Процесс осуществления привязки можно обсуждать в двух аспектах: временном и информационном. Временной аспект характеризует период времени между написанием программы и выполнением отдельного оператора этой программы. Информационный аспект отображает типы информации, используемые при выполнении привязки, и включение описания параметров данных, их значений и местоположения.

В случае привязки конкретного оператора программы к типовой вычислительной системе временной аспект может охватывать следующие моменты: 1) привязку в процессе написания оператора; 2) привязку в процессе компилирования; 3) привязку в процессе установления связей с подпрограммами (например, на этапе, называемом «редактирование и установление связей»); 4) привязку в процессе загрузки программы; 5) привязку в процессе вызова подпрограммы; 6) привязку в процессе выполнения предыдущего оператора программы; 7) привязку в процессе выполнения данного оператора.

Привязка в процессе написания оператора предполагает принятие решений, которые в дальнейшем не могут быть изменены. Например, при записи оператора

$A := A + 3;$

осуществлена привязка к значению 3. Если же используется оператор

$$A := A + N;$$

то указанной выше привязки не происходит, значение переменной N в дальнейшем можно изменять. Если в процессе написания программы программист включает в ее текст информацию, зависящую от специфических характеристик машины, то происходит привязка программы к машине. Одним из главных отличий программ на языках высокого уровня от программ на языках ассемблера является значительно большая степень привязки последних к машине уже на этапе их написания.

В большинстве традиционных вычислительных систем привязка программы к машине осуществляется в значительной степени на этапе компилирования, когда идентификаторы переменных преобразуются в адреса, а операторы программы — в последовательности машинных команд. В машине с традиционной архитектурой фон Неймана привязка всех атрибутов (определителей данных) обычно выполняется именно на этом этапе. Например, при компилировании оператора

$$A := A + 1$$

компилятор принимает решение, каким — двоичным, с плавающей точкой или десятичным — должно быть сложение и какова точность этой операции. В первых вычислительных системах компиляторы создавали абсолютные (неперемещаемые) объектные программы, предопределяя тем самым привязку местоположения программы (назначение программе конкретных адресов памяти) на этапе компилирования.

Системы с отдельным этапом установления связей с подпрограммами предполагают осуществление определенного вида привязки не ранее выполнения этого этапа. Например, подпрограмма может содержать обращение к другой подпрограмме X , и привязка первой подпрограммы к упомянутой подпрограмме X может быть отнесена на время, следующее за этапом компилирования.

Дальнейшая привязка, обычно связанная с указанием местоположения, происходит во время загрузки программы. В зависимости от специфики вычислительной системы на этапе загрузки программы может происходить полная привязка последней к определенной области памяти без возможности дальнейшего ее перемещения в памяти или частичная привязка, допускающая последующее перемещение.

В зависимости от специфики программы, вычислительной системы и языка программирования привязка определенного вида может быть отложена до тех пор, пока в процессе выпол-

нения программы не произойдет вызов требуемой подпрограммы. Если в языке программирования предусмотрена возможность использования локальных переменных, местоположение последних устанавливается как раз в это время. При передаче параметров в подпрограмму их привязка к конкретным значениям осуществляется именно на этапе вызова подпрограммы. (Некоторые особенности средств передачи параметров в подпрограмму, такие, как передача параметров по именн, позволяют отложить привязку рассматриваемого вида даже на более поздний период.)

Ряд параметров программы делает возможным отложить привязку определенного вида до момента выполнения первого оператора программы. Например, оператор

$A(I) := B;$

оказывается связан с определенным адресом области памяти только после выполнения некоторого предшествующего оператора, задающего значение переменной I . В определении

$BUFFER : \text{access STRING};$

языка Ада местоположение цепочки $STRING$ и ее размер оказываются заданными (привязанными к вычислительной системе) только после того, как некоторый оператор присвоит переменной $BUFFER$ определенное значение.

Наконец, все привязки, имеющие отношение к некоторому оператору, не выполненные на предыдущих этапах, подлежат реализации в процессе выполнения этого оператора программы.

Итак, многие свойства современных языков программирования, операционных систем, архитектуры вычислительных машин отдалают процесс привязки программы к машине на все более поздние этапы подготовки программы и ее выполнения. Как упоминалось выше, при использовании языков высокого уровня объем работ по привязке, выполняемой программистом на этапе написания программы, сокращается. Применение индексных регистров и косвенной адресации позволяет отложить привязку команды к объекту с конкретным местоположением до начала выполнения программы. Использование в языке ПЛ/1 символа «звездочка» для указания длины строки или размерности переменной, а в языке Ада нелимитированных формальных параметров дает возможность отложить процесс привязки операторов программы к определенным значениям аргументов подпрограммы до момента обращения к ней. Виртуальная память позволяет осуществлять привязку ее адресов к физическим адресам операндов команды не ранее начала выполнения последней. Процедура открытия файла предоставляет возможность не наз-

начать программе конкретный файл вплоть до начала ее выполнения.

Итак, наблюдается тенденция к задержке процесса привязки программ к конкретным описаниям параметров данных, их значений и местоположения. Главное достоинство этого явления состоит в определенной гибкости работы программиста, благодаря чему программа носит более обобщенный характер в течение большего периода времени. (Исключением является язык Ада, в котором значительный объем операций по привязке перенесен обратно на этап компилирования, поскольку в процессе разработки языка было сделано заключение о том, что машины с традиционной архитектурой не обеспечивают в достаточной степени сохранность программ.)

Согласно изложенному в гл. 2, архитектура машины фон Неймана способствует перемещению основного объема работ по привязке программы к машине на этап компилирования (например, это определяется как «абсорбирование структуры и характеристик данных в логику работы программы»). Отсюда следует, что усилия по сокращению семантического разрыва между языком программирования и машиной требуют формулировки принципов переноса части или всего объема работ по привязке программы к машине на более поздние этапы, вплоть до этапа выполнения программы. Прежде чем перейти к детальному рассмотрению этих принципов (гл. 4), обсудим в данной главе некоторые основные подходы к решению проблемы упомянутого семантического разрыва, приведя по каждому из этих подходов соответствующие литературные источники. Данный обзор ограничен только проблемой семантического разрыва между языком программирования и машиной.

Рис. 3.1 может помочь сформулировать некоторые из основных подходов к определению архитектуры вычислительной машины. Ветвь 1 символизирует традиционный подход: посредством обширного процесса компилирования программа на языке высокого уровня переводится в программу на машинном языке, которая затем интерпретируется машиной. Первая разновидность архитектуры, ориентированной на сокращение семантического разрыва между языком программирования и машиной, представлена ветвью 2: исходная программа компилируется в программу на машинном языке более высокого уровня; последняя подлежит интерпретации машиной. Архитектура вычислительных систем такого типа известна под названием *архитектуры, ориентированной на язык высокого уровня*.

Ветви 3 и 4 на рис. 3.1 символически представляют три архитектурных решения; соответствующие вычислительные машины имеют общее наименование *машин языков высокого уровня*. Машины с архитектурой, представленной ветвью 3, достигают

такого уровня, когда язык высокого уровня можно рассматривать как язык ассемблера. Говоря иначе, есть взаимно-однозначное соответствие типов операторов и знаков операций языка высокого уровня с командами машинного языка. В данном случае правильно говорить об *ассемблировании*, а не о компи-

А - ассемблируемая программа
К - компилируемая программа
И - интерпретируемая программа

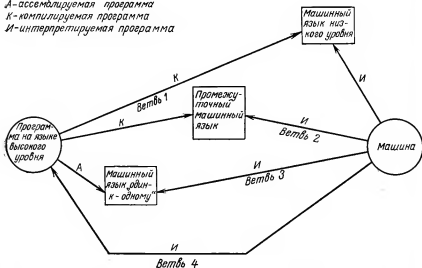


Рис. 3.1. Взаимосвязь между программами на языке высокого уровня и на машинном языке.

лировании исходной программы. Ветвь 3 представляет две разновидности архитектуры, в одной из которых ассемблирование выполняется программными, а в другой аппаратными средствами. Архитектура, соответствующая ветви 4, отличается тем, что здесь язык высокого уровня является также и машинным языком; машина интерпретирует непосредственно программу на языке высокого уровня.

АРХИТЕКТУРА, ОРИЕНТИРОВАННАЯ НА ЯЗЫК ВЫСОКОГО УРОВНЯ

Речь идет о машинах, организация данных и набор операций которых по структуре значительно теснее связаны с организацией данных и набором операций одного или нескольких языков высокого уровня. Указанное выше название архитектуры таких машин не является ни строгим по определению, ни общепринятым, что, впрочем, характерно для большинства терминов

вычислительной техники. Например, если к набору команд Системы 370 добавить команду, семантически подобную оператору языка КОБОЛ PERFORM-UNTIL, архитектура этой вычислительной системы стала бы в большей степени ориентирована на язык программирования, однако этого недостаточно для классификации ее как архитектуры, ориентированной на язык высокого уровня. Последняя предполагает, что соответствующая ей машина проектируется с оказанием требованиям одного или нескольких языков высокого уровня явного приоритета перед другими возможными требованиями.

Термин «архитектура, ориентированная на язык высокого уровня» впервые, по-видимому, ввел У. Мак-Кимен [2], хотя архитектуру этого типа анализировали и другие ученые [3, 4]. К вычислительным системам, ориентированным, например, на АЛГОЛ, относятся системы B5500/6500/6700/7600 фирмы Burroughs [5, 77], KDF.9 [6] и MU5 [7]. Предпринимались также попытки разработать архитектуру, в которой были бы отражены некоторые общие характеристики языков с блочной структурой [3, 8—11]. Разработана машина с архитектурой, ориентированной на язык TPL с блочной структурой и операциями, подобными тем, которыми располагает язык APL [12].

L-машина имеет архитектуру с традиционной организацией памяти, однако набор ее команд ориентирован на язык ПЛ/1 [13, 14]. Эта архитектура была реализована на машине с микропрограммным управлением Microdata 1621. Пример другой машины, ориентированной на язык ПЛ/1, описан в работе Сугimoto [15]. На диалект подмножества языка ПЛ/1 ориентирована учебная ПЛ-машина, рассматриваемая во второй части данной книги [16].

ЭВМ B3500 фирмы Burroughs является ориентированной на КОБОЛ машиной, предназначенной для финансово-экономических расчетов; фирма NCR Criterion недавно разработала два архитектурных решения машины: традиционное и ориентированное на КОБОЛ [17, 18]. Переход от одной архитектуры к другой выполняется динамически, под управлением средств взаимосвязи подпрограмм. Исследования показывают, что программы на КОБОЛе, компилируемые в расчете на архитектуру, ориентированную на КОБОЛ, выполняются в четыре раза быстрее, чем в случае их компилирования для традиционной архитектуры. В литературе [19—21, 78, 80] описывается несколько экспериментальных машин, архитектура которых также ориентирована на КОБОЛ.

Поскольку в настоящее время вызывает интерес язык Паскаль, не удивительно, что появляются экспериментальные машины с архитектурой, ориентированной на этот язык [22, 79—81]. Одна из них выполнена на микропроцессорах, производ-

ных фирмой Western Digital. То же самое можно сказать и в отношении языка ЛИСП. В Массачусетском технологическом институте разработан проект машины CONS с архитектурой, ориентированной на язык ЛИСП [23, 24]; машина предназначена для использования в том же институте. Кроме того, было разработано по крайней мере два варианта машин с архитектурой подобной ориентации [25—27]. Несколько небольших фирм сообщили о том, что располагают возможностями начать с 1981 г. производство машин с архитектурой, ориентированной на язык ЛИСП, которые предназначены для ведения финансово-экономических расчетов.

Для создания машин языков высокого уровня и машин, ориентированных на языки программирования, широко используется язык APL. Так, машина Raytheon AADC ориентирована на язык APL [28, 29]. То же можно сказать и о ЭВМ STARLET с экспериментальной архитектурой, располагающей мощными средствами по структурированной организации данных [30, 31]. Авторы работы [32] сообщают об архитектуре, предоставляющей возможность управления программными средствами конфигурацией аппаратных ресурсов вычислительной системы. Эта архитектура предусматривает набор операций, подобный тому, которым располагает язык APL. Опубликованы сведения о проектах машин с архитектурой, ориентированной на языки Balm [33], Mary [34], Snobol [35], на расширенную версию языка CLU [36], не имеющий названия алгоритмический язык [37], язык HLAX [82] и язык реального времени [83].

Иным подходом к созданию машин подобной архитектуры является ориентация последней не на один конкретный язык, а на общие семантические характеристики некоторой группы языков. Примерами подобных решений могут служить вычислительные машины Rice Research Computer [38], BLM [39] и др. [40—44]. Разработчики машины SWARD [45], описываемой в части V книги, не стремились к созданию архитектуры, ориентированной на определенный язык высокого уровня, а пытались построить машину, обеспечивающую программисту благоприятную рабочую среду. В результате таких усилий была сконструирована машина с архитектурой, ориентированной на языки Ада, ПЛ/1, Фортран, Кобол и т. п. Другим примером такого подхода является машина iAPX 432 фирмы Intel, рассматриваемая в части VI книги. Архитектура этой машины до некоторой степени ориентирована на язык Ада.

Промежуточное положение между двумя указанными подходами к ориентации архитектуры машины на языки программирования (ориентация на конкретный язык или на обобщенные свойства языков) занимает описываемая в части IV архитектура машины B1700 фирмы Burroughs. В процессе работы машины

посредством вызова различных микропрограмм можно динамически переходить от одного набора команд к другому, ориентируя тем самым архитектуру машины на тот язык, на котором написана подлежащая выполнению программа.

АРХИТЕКТУРА МАШИН ЯЗЫКОВ ВЫСОКОГО УРОВНЯ. ТИП А

Теперь перейдем к рассмотрению архитектуры, соответствующей ветви 3 на рис. 3.1. В указанном случае архитектура машины должна быть ориентирована на язык высокого уровня до такой степени, когда последний воспринимается как язык ассемблера (машинный язык в символической форме). Иначе говоря, при этом должно быть однозначное соответствие между типами операторов и знаками операций языка высокого уровня, с одной стороны, и машинными операциями — с другой. Тогда программа, преобразующая исходную программу в программу на машинном языке и обычно называемая компилятором, получает более точное наименование — *ассемблер*. Она убирает из исходной программы комментарии и пробелы, преобразует знаки операций, разделители и ключевые слова в более короткие машинные коды, имена переменных — в адреса, реорганизует, если это необходимо, выражения из инфиксной формы записи в префиксную, но не выполняет традиционных функций компилятора. Следовательно, при таком подходе к решению проблемы большая часть объема работ по привязке программы к машине откладывается до начала выполнения программы.

Хотя описываемый подход и классифицируется как второе из возможных решений в рамках традиционной архитектуры фон Неймана, невозможно провести четкую грань, отделяющую этот подход от подхода, при котором архитектура ориентирована на язык высокого уровня. Имеющие место различия носят скорее количественный характер: архитектура типа А машин языков высокого уровня может рассматриваться как архитектура с высокой степенью ориентации на языки высокого уровня.

Примерами машин с архитектурой типа А являются системы реального времени для языка FLUID [46], бортовая вычислительная система для языка Space Programming Language (являющегося производным от языка Jovial) [47], ФОРТРАН-машина [48], АЛГОЛ-W-машина [49], вычислительная система для языка ISPL [50], ЛИСП-микропроцессор [51], другие ЛИСП-машины [80, 85] и APL-машина [52]. Кроме этого, для ЭВМ модели 25 Системы 360 были разработаны микропрограммы, генерирующие APL-машину [53], а наличие на некоторых моделях Системы 370 специальных средств, называемых APL

Assist, предоставляет в распоряжение пользователей микропрограмму для интерпретации подвергнутых ассемблированию APL-программ [54]. Отметим также, что внутренняя организация процессора Interpreter фирмы Burroughs [55] спроектирована таким образом, что позволяет создавать на своей основе машины языков высокого уровня с архитектурой типа А.

АРХИТЕКТУРА МАШИН ЯЗЫКОВ ВЫСОКОГО УРОВНЯ. ТИП Б

Архитектура этого типа почти идентична рассмотренной выше и тоже представлена ветвью 3 на рис. 3.1. Отличие архитектуры типов А и Б состоит в том, что в машине с архитектурой типа А процесс ассемблирования выполняется программным путем, а в машине с архитектурой типа Б—аппаратным или микропрограммным путем, т. е. в последнем случае машина сначала ассемблирует исходную программу, а затем интерпретирует. Примерами машин с архитектурой типа Б могут служить APL-машины [56—58], ФОРТРАН-машина [59], Эйлер-машинна [60], БЕЙСИК-машина [61], машина, ориентированная на синтаксис языка [62], АЛГОЛ-машина [63] и, наконец, система SYMBOL, рассматриваемая в части III данной книги [64].

Отметим, что машинам с архитектурой типа Б присущ тот же семантический разрыв между языком и машиной, который характеризует машины с архитектурой типа А. Единственным достоинством машин с архитектурой типа Б является большая скорость процесса ассемблирования, поскольку он реализуется аппаратно или микропрограммно; однако для большинства практических применений это достоинство не является существенным. К недостаткам машин с архитектурой типа Б относятся более высокая стоимость и меньшие возможности по расширению (перестройке). Это связано с более высокой стоимостью реализации тех или иных функций (а также любых изменений этих функций) микропрограммными или аппаратными средствами по сравнению с программным способом. Разработчики вычислительных машин часто руководствуются следующими критериями при принятии решения о реализации тех или иных функций с помощью микропрограммных или аппаратных средств: во-первых, функция должна быть сравнительно простой; во-вторых, должна быть маловероятной потребность ее изменения в будущем; в-третьих, должна быть крайне невыгодна меньшая скорость выполнения этой функции в случае ее реализации программными средствами. Остается неясным, удовлетворяют ли какому-нибудь из этих критериев функции ассемблера.

СООТНОШЕНИЯ МЕЖДУ ПОКАЗАТЕЛЯМИ СТОИМОСТИ АППАРАТНОЙ И ПРОГРАММНОЙ РЕАЛИЗАЦИИ ФУНКЦИИ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

Ввиду важности учета экономического фактора сделаем небольшое отступление и рассмотрим соотношения между стоимостными показателями аппаратной и программной реализации тех или иных функций (относящихся как к архитектуре машины, так и к программному обеспечению последней).

Если некоторая требуемая функция исключена из списка функций, реализуемых аппаратными средствами машины, она подлежит выполнению программным путем. Если же она реализуется аппаратно, это связано с работой определенных комбинационных и последовательных схем и (или) микропрограммируемых логических схем. Чтобы сопоставить стоимость указанных альтернативных решений, необходимо принять во внимание затраты на разработку и производство соответствующих средств. Общеизвестной является высокая стоимость разработки программного обеспечения. Иногда полагают, что его создание не связано с производственными затратами. Однако, чтобы убедиться в обратном, достаточно принять во внимание затраты на производство запоминающих устройств, необходимых для размещения программного обеспечения.

По сравнению с реализацией функции программными средствами использование микропрограммных средств связано с более высокими затратами на разработку и производство. Факт высоких затрат на разработку является общепризнанным и не требует дальнейших обсуждений. Возрастание производственных расходов объясняется следующими причинами: во-первых, традиционное программирование в отличие от микропрограммирования характеризуется более высоким уровнем алгоритмизации, и вследствие этого требуется меньший объем памяти для результирующей программы; во-вторых, микропрограммы обычно хранятся в более быстродействующей, а следовательно, и более дорогостоящей запоминающей среде. По сравнению с реализацией функции программными и микропрограммными средствами использование для этих целей логических схем центрального процессора сопряжено с еще более значительными затратами на их разработку и производство.

Все предшествующие рассуждения базировались на том, что реализация тех или иных функций вычислительной системы программными средствами является всегда несколько менее дорогостоящей. Однако необходимо принять во внимание еще два обстоятельства. Во-первых, изъятие какой-либо функции у машины может предполагать ее многократную реализацию про-

граммными средствами (например, всеми компиляторами). Так, если в машине отсутствуют средства вызова подпрограмм, в компиляторе каждого языка программирования (и, возможно, в ряде прикладных программ) необходимо предусмотреть реализацию этих средств программным путем. Хотя однократная программная реализация этой функции системы может оказаться и дешевле адекватных аппаратных средств, общая стоимость всего соответствующего программного обеспечения может быть дороже. Во-вторых, необходимо принять во внимание особенности затрат на производство программных средств. Отсутствующая в машине и реализуемая программно та или иная функция системы может оказаться многократно дублируемой в памяти машины. Предположим, что в машине не предусмотрена возможность адресации к элементам массивов или преобразования данных из одной формы представления в другую. Тогда при каждом «прочтении» компилятором обращения к массиву (или запроса на преобразование данных) компилятор должен генерировать, например, восемь команд для выполнения требуемой операции. Эти восемь команд появляются в памяти неоднократно (при каждом обращении к массиву или запросе на преобразование данных). Последнее обстоятельство увеличивает суммарную стоимость производства программных средств.

Для более детального анализа влияния указанных факторов сравним затраты на аппаратную и программную реализации некоторой функции в одной и той же системе. Рассмотрим такую функцию системы, реализация которой в машине требует и аппаратных и микропрограммных средств. Если D_h и M_h — стоимость разработки и производства аппаратных средств реализации требуемой функции, а V — количество производимых вычислительных систем с этой функцией, то стоимость аппаратной реализации этой функции в одной системе равна $D_h/V + M_h$. (Известно, что M_h меняется с изменением значения V , но для простоты это обстоятельство принимать во внимание не будем.)

Пусть D_s — стоимость разработки программных средств реализации той же функции, M_s — стоимость производства этих средств (например, стоимость памяти, необходимой для хранения соответствующего фрагмента программы), C — число раз генерирования этой функции в форме фрагмента программы, R — число раз, которое запрограммированная функция встречается в памяти в процессе работы системы. Тогда стоимость программной реализации рассматриваемой функции в одной системе равна $C \times D_s/V + R \times M_s$.

В результате имеем уравнение с семью неизвестными. Однако для большинства вычислительных систем $D_h \gg M_h$ (предельный случай относится к микропроцессорам). Поскольку ранее указывалось, что $D_s < D_h$, а $M_s < M_h$, то справедливы следующие

щие отношения: $D_h > D_s > M_h > M_s$. Согласно рыночным ценам 1981 г., стоимость «производства» одного оператора программы равна ~ 100 долл.; при этом стоимость 16К бит памяти составляет ~ 1 долл. Полагая, что оператор языка высокого уровня, представленный в машинном коде, занимает 20 байт памяти ЭВМ стандартной архитектуры, получим $D_s = 10\,000 \times M_s$. В целях конкретизации дальнейших рассуждений положим также, что операнды указанных выше неравенств — члены геометрической прогрессии, знаменатель которой есть корень квадратный из 10 000, т. е. $D_h = 100 \times D_s$; $D_s = 100 \times M_h$ и т. д. (Отметим, что эти соотношения — результат умозрительных заключений, которые выглядят весьма правдоподобными.)

Тогда стоимость затрат на аппаратную реализацию равна $1\,000\,000 \times M_s/V + 100 \times M_s$, а на программную реализацию составляет $10\,000 \times C \times M_s/V + R + M_s$. Следовательно, если $1\,000\,000/V + 100 < 10\,000 \times C/V + R$, то затраты на аппаратную реализацию меньше. Если $V=1$ (производится только один комплект вычислительной системы), а величина переменной C большая, то стоимость аппаратных средств меньше, чем программных, при $C > 100$ (программные средства реализации требуемой функции дублируются более 100 раз). Подобная ситуация возможна, но представляется маловероятной. Для многих функций, подлежащих реализации, ситуацией, имеющей большую вероятность, можно считать такую, при которой $V=10\,000$, а параметры C и R представляют собой значительную величину; при этом аппаратная реализация дешевле, если $C+R > 200$. Весьма вероятна также ситуация, когда при $V=1\,000\,000$ (например, организовано производство микропроцессора) аппаратная реализация дешевле, если $R > 101$. Итак, аппаратная реализация экономически оправдана при следующих обстоятельствах: во-первых, при условии производства большого количества экземпляров проектируемой системы; во-вторых, если требуется многократное генерирование адекватной программной реализации; в-третьих, если в памяти машины оказывается большое число дубликатов программной реализации требуемой функции. Ситуации первого типа обычно возникают при работе с микропроцессорными наборами, две другие — при использовании больших вычислительных систем в мультипрограммном режиме.

Теперь мы пришли к пониманию того, почему экономически неоправданно практическое использование машин языков высокого уровня с архитектурой типа Б. Программная реализация процесса ассемблирования (компилирования), вероятнее всего, создала бы ситуацию, при которой $C=R=1$; в результате условия для разделения функций между аппаратными и программными средствами отсутствовали бы; аппаратная реализация оказалась бы более дорогой.

В заключение следует отметить, что показатель стоимости не является единственным критерием в поиске компромисса при распределении функций системы между ее аппаратными средствами и программным обеспечением. Не менее важным является такой показатель, как быстродействие. Поскольку быстродействие аппаратных средств всегда выше, приведенные формулы подлежат пересмотру с позиций поиска компромисса между стоимостью тех или иных средств и их быстродействием. Очевидно, что в ситуациях приблизительно равной стоимости аппаратной и программной реализаций предпочтение будет отдаваться первой. Немаловажными факторами в поиске упомянутого выше компромисса являются защита системы от несанкционированного доступа и функциональная целостность. Если этим факторам уделяется значительное внимание, то независимо от требований экономии средств и достижения более высокого быстродействия предпочтение будет отдано аппаратным средствам.

АРХИТЕКТУРА МАШИН ЯЗЫКОВ ВЫСОКОГО УРОВНЯ. ТИП В

Очевидным решением проблемы полного устранения семантического разрыва между языком программирования и машиной и перенесением процесса привязки программы к машине на этапе выполнения программ является создание машины, непосредственно интерпретирующей язык высокого уровня. На рис. 3.1 это решение символически обозначено ветвью 4. Здесь стираются различия между понятиями архитектуры машины и языка программирования; они становятся идентичными. Примерами машин с подобной архитектурой являются APL-машина [65], АЛГОЛ-машина [66—68], БЕЙСИК-машина [80], ФОРТРАН-машина [85], Паскаль-машина [86], ЛИСП-машина [71, 80], Jovial-машина [87], машины языков Adam [69], IPL [70], LAX [72], машина языка, не имеющего названия [74], и машина со структурой, подстраиваемой под любой язык [75]. Машины с архитектурой подобного типа встречаются не только в лабораторном (экспериментальном), но и в промышленном исполнении. Примером машины последнего вида является портативная вычислительная машина 5100/5110 фирмы IBM. Когда машина, подобная ЭВМ модели 5100, интерпретирует программы на языке APL, архитектура ее процессора представляет собой разновидность архитектуры Системы 370. Этот процессор интерпретирует программно-реализованный интерпретатор языка APL, который в свою очередь интерпретирует программу на языке APL. То же происходит, когда этот процессор интерпретирует программы на БЕЙСИКе, с той лишь разницей,

что в этом случае процессор загружается набором команд, подобным набору команд вычислительной системы S/3 фирмы IBM, и интерпретирует программно-реализованный интерпретатор языка БЕЙСИК.

Хотя архитектура типа В и сводит семантический разрыв к нулю, ее недостатки представляются столь значительными, что архитектуру такого типа вряд ли можно считать перспективной [84]. Прежде всего не ясно, можно ли вообще в данном случае говорить об архитектуре машины. Процесс создания архитектуры вычислительной машины предполагает поиск компромисса в распределении ее функций между аппаратными средствами и программным обеспечением. Что же касается рассматриваемой архитектуры, то здесь выполнение всех функций возлагается на аппаратные средства. Относительно редко используемых функций или таких, скорость реализации которых не является важной характеристикой, разработчик принимает решения, которые далеки от оптимальных.

Машины с архитектурой типа В должны быть также менее эффективны, чем машины с архитектурой ранее рассмотренных типов, прежде всего потому, что вся работа по привязке программы к машине осуществляется в процессе прогона программы динамически, а следовательно, методом повторений. При каждом выполнении машиной оператора программы должны производиться его лексический анализ, синтаксический разбор этого оператора, преобразование символических имен в адреса и т. п. Например, если конкретный оператор IF используется 1000 раз, то столько же раз осуществляется его синтаксический разбор. В то же время в машинах с архитектурой другого типа для каждого оператора указание выше действия производится компилятором или ассемблером только один раз. Исключительная неэффективность машины с архитектурой типа В объясняется необходимостью манипуляции символическими именами переменной длины, оперирования выражениями в инфиксной форме и сканирования программы в поиске точек перехода (например, при обработке операторов IF и обращений к подпрограммам). Высказывались разнообразные предложения [68, 76] по устранению этих недостатков, однако сделанные предложения являются, по существу, платой за сложность исходных решений.

По сравнению с машинами выше рассмотренных архитектурных решений машины с архитектурой типа В по тем же причинам, что и машины с архитектурой типа Б, отличаются более высокой стоимостью и меньшей способностью к перестройке и расширению своих функциональных возможностей.

Еще одним недостатком машины с архитектурой типа В является отсутствие процедуры (типа компилирования или ас-

семблирования), осуществляющей анализ программы до ее выполнения. Очень часто забывают об одном очень важном достоинстве компилятора или ассемблера — способности обихуживать в программе синтаксические ошибки. В машине данного типа ошибки не выявляются вплоть до этапа выполнения программы (например, синтаксическая ошибка в конкретном операторе может оставаться незамеченной даже по истечении 30 мин выполнения программы). Конечно, можно создать предпроцессор для анализа синтаксических ошибок, однако это повлекло бы дополнительные расходы; пришлось бы двукратно производить синтаксический анализ программы: предпроцессора и непосредственно в ходе выполнения машиной программы. Конечно, в таком случае предпроцессор мог бы преобразовывать исходную программу в форму, более удобную для последующего выполнения. Однако в результате машина с архитектурой типа В была бы превращена в машину с архитектурой типа А или Б.

Заметим, наконец, что проекты построения машины с архитектурой типа В часто упускают из виду реализацию таких функций (являющихся скорее функциями системы, а не языка программирования), как ввод-вывод, управление вычислительным процессом и управление ресурсами памяти.

В заключение укажем, что в дальнейшем в данной книге архитектура типа В рассматриваться не будет, поскольку при формулировании принципов ее построения не учитывались тот факт, что некоторые функции системы лучше реализовать программными средствами, а также то обстоятельство, что наилучшим решением является преобразование исходной программы на этапе, предшествующем ее выполнению.

АРХИТЕКТУРА МАШИНЫ И КОМПИЛЯТОРЫ

Простота компилирования программ — один из доводов в пользу построения машины с архитектурой, ориентированной на язык программирования. Существует мнение, что «главным достоинством той или иной машины с архитектурой, ориентированной на язык Паскаль, является упрощение процесса разработки компилятора», и, кроме того, «стеки имеют преимущество перед регистрами в том, что упрощают формирование кодов для разработчика компилятора». К утверждениям подобного рода следует относиться критически, поскольку они часто носят дезинформирующий характер.

В общем случае разработчику вычислительной системы не следует обращать чрезмерное внимание на те последствия, которые оказывают на компилятор формулировка принципов по-

строения и принятие решений об организации архитектуры системы. В настоящее время типичные вычислительные системы (микропроцессорные, ЭВМ индивидуального пользования, небольшие системы для делопроизводства, малые и сверхбольшие ЭВМ) могут быть оснащены одним или несколькими компиляторами, операционной системой или управляющей программой и другими программными средствами, поставляемыми производителями машин (библиотеками подпрограмм, программами-утилитами, лицензионными прикладными программами). Повышенное внимание к проблеме снижения стоимости разработки компилятора окажется необоснованным, если принять во внимание весь комплекс усилий, затрачиваемых на разработку исходного объема аппаратных средств и программного обеспечения проектируемой системы, а также учесть все те программы (возможное количество которых измеряется тысячами), которые предназначены для системы и подлежат разработке в последующий период. Большой ущерб может нанести пренебрежение каким-либо более важным фактором.

Как уже неоднократно упоминалось, перед разработчиком архитектуры вычислительной системы стоит задача компромиссного распределения функций системы между аппаратными средствами и программным обеспечением, оптимизация этого распределения с учетом их стоимости, анализа функционирования системы на микро- и макроуровнях, проблем модификации системы, ее надежности и т. д. При анализе функционирования системы на макроуровне особое внимание уделяется влиянию выбора варианта построения архитектуры на стоимость соответствующего программного обеспечения. Как известно, компилятор — это только часть программного обеспечения, которая создается лишь один раз, а потребность в написании прикладных программ может возникать ежедневно. Принимая во внимание это обстоятельство, а также то, что специализированные программы, как правило, разрабатываются соответствующими специалистами, трудно найти достаточно убедительные доводы в пользу выбора варианта построения архитектуры системы, руководствуясь прежде всего желанием минимизировать затраты на разработку компилятора. При анализе функционирования системы на микроуровне также не представляется обоснованным стремление к увеличению скорости процесса генерирования компилятором машинных кодов. В конечном счете вычислительная система создается не как машина для компилирования программ, а как средство эффективного выполнения программ, прошедших этап компилирования (что, возможно, имело место несколько месяцев или лет тому назад).

Проблемы, связанные с процессом компилирования, не имеют универсальных решений, как, впрочем, и большинство дру-

гих проблем архитектуры вычислительных машин. Однако существуют по крайней мере две ситуации, в которых от разработчика архитектуры ЭВМ требуется особое внимание к процессу компилирования: 1) система вынуждена затрачивать необычно много времени на компилирование (например, система проектирования микропроцессоров); 2) архитектура системы препятствует настройке компилятора на оптимизацию порождаемых им машинных кодов. В первом случае, встречающемся довольно редко, у разработчика архитектуры ЭВМ имеются достаточные основания быть озабоченным проблемой скорости компилирования, хотя ее решение вовсе не обязательно связано с увеличением затрат на разработку компилятора. Во втором случае влияние организации вычислительной системы в целом на успех решения проблемы очевидно. Однако это обстоятельство может оказаться не столь значительным, поскольку большая часть работы по оптимизации может быть выполнена без учета специфики конкретного архитектурного решения.

В заключение отметим, что нет ничего принципиально ошибочного в стремлении добиться от разработчика архитектуры ЭВМ определенных решений, упрощающих компилятор или облегчающих труд по его написанию; все зависит от того, чему именно уделяется наибольшее внимание. Если при решении основных проблем архитектуры машины попутно удается упростить компилятор и увеличить скорость его работы, это можно только приветствовать. Однако к этому следует относиться как к полезному побочному эффекту, но не цели проектирования архитектуры.

УПРАЖНЕНИЯ

3.1. Одной из опасностей попыток сокращения семантического разрыва между машиной и данным языком программирования является возможное увеличение семантического разрыва между этой машиной и другим языком программирования. Сформулируйте несколько подходов, используя которые можно решить эту проблему.

3.2. Учитывая сравнительную всеобъемлемость архитектуры машин языков высокого уровня и архитектуры, ориентированной на подобные языки, укажите, какие языки «выпали из поля зрения» этих архитектурных решений?

ЛИТЕРАТУРА

1. Radin G., A Note on the Concept of Binding, RC-3287, IBM Research Div., Yorktown Heights, NY, 1971.
2. McKeeman W. M., Language Directed Computer Design, Proceedings of the 1967 Fall Joint Computer Conference, Washington, Thompson, 1967, pp. 413—417.
3. McMahan L. N., Feustel E. A., Implementation of a Tagged Architecture for Block Structured Languages, Proceedings of the ACM-IEEE Symposium on

- High-Level Language Computer Architecture, New York, ACM, 1973, pp. 91—100.
4. Brooker R. A. Influence of High-Level Languages on Computer Design, Proceedings of the IEE, 117(7), 1219—1224 (1970).
 5. Hauck E. A., Dent B. A. Burroughs B6500/7500 Stack Mechanism, Proceedings of the 1968 Spring Joint Computer Conference, Washington, Thompson, 1968, pp. 245—251.
 6. Haley A. C. D., The KDF.9 Computer System, Proceedings of the 1962 Fall Joint Computer Conference, Washington, Spartan, 1962, pp. 108—120.
 7. Ibbett R. N., Capon P. C., The Development of the MU5 Computer System, Communications of the ACM, 21(1), 13—24 (1978).
 8. Miller J. S., Vandever W. H., Instruction Architecture of an Aerospace Multiprocessor, Proceedings of the ACM-IEEE Symposium on High-Level Language Computer Architecture, New York, ACM, 1973, pp. 52—60.
 9. Lutz M. J., Manthey M. J., A Microprogrammed Implementation of a Block Structured Architecture, Record of the Fifth Annual Workshop on Microprogramming, New York, ACM, 1972, pp. 28—41.
 10. Kilburn T., Morris D., Rohl J. S., Sumner F. H., A System Design Proposal, Proceedings of the IFIP Congress 1968, Amsterdam, North-Holland, 1969, pp. 806—811.
 11. Lindsey C. H., Making the Hardware Suit the Language, in J. E. L. Peck, Ed., ALGOL 68 Implementation, Amsterdam, North-Holland, 1971, pp. 347—365.
 12. McFarland C., A Language-Oriented Computer Design, Proceedings of the 1970 Fall Joint Computer Conference, Montvale, NJ, AFIPS, 1970, pp. 629—640.
 13. Wade B. W., Schneider V. B., A General-Purpose High-Level Language Machine for Minicomputers, Proceedings of the ACM SIGPLAN-SIGMICRO Interface Meeting, New York, ACM, 1973, pp. 169—171.
 14. Wade B. W., Schneider V. B., The L-Machine: A Computer Instruction Set for the Efficient Execution of High-Level Language Programs, Record of the Fifth Annual Workshop on Microprogramming, New York, ACM, 1972, pp. 81—82.
 15. Sugimoto M., PL/1 Reducer and Direct Processor, Proceedings of the 24th ACM National Conference, New York, ACM, 1969, 519—538.
 16. Wortman D. B., A Study of Language Directed Computer Design, Ph. D. dissertation, Stanford University, Stanford, CA, 1972.
 17. Tang T., O'Flaherty K., Virtual Machines and the NCR Criterion, *Datamation*, 24(4), 129—134 (1978).
 18. Shapiro M. D., The Criterion Cobol System, Proceedings of the 1978 NCC, Montvale, NJ, AFIPS, 1978, pp. 1049—1054.
 19. Chevance R. J., A COBOL Machine, Proceedings of the ACM SIGPLAN-SIGMICRO Interface Meeting, New York, ACM, 1973, pp. 139—144.
 20. Illiffe J. K., Interpretive Machines, TR-149, Digital Systems Laboratory, Stanford University, Stanford, CA, 1977.
 21. Yamamoto M., Hakoziaki K., A Cobol Oriented High Level Language Machine, *EURMICRO Newsletter*, 3(4), 58—64 (1977).
 22. Schoellkopf J. P., Baille G., PASC-HLL, A Pipelined Architecture Bit-Slice Computer for High Level Language, Digest of Papers, Fall Comcon77, New York, IEEE, 1977, pp. 46—51.
 23. Schoichet S. R., The LISP Machine, *Mini-Micro Systems*, 11(5), 68—74 (1978).
 24. Bawden A. et al., LISP Machine Progress Report, AIM-444, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1977.
 25. Deutsch L. P., Experience with a Microprogrammed Interlisp System, Micro-11 Proceedings, New York, ACM, 1978, pp. 128—129.
 26. Griss M. L., Kessler R. R., REDUCE/1700, A Micro-Coded Algebra System, Micro-11 Proceedings, New York, ACM, 1978, pp. 130—138.

27. Griss M. L., Swanson M. R., MBALM/1700, A Microprogrammed LISP Machine for the Burroughs B1726, Micro10 Proceedings, New York, ACM, 1977, pp. 15—25.
28. Nissen S. M., Wallach S. J., The All Applications Digital Computer, Proceedings of the ACM-IEEE Symposium on High-Level Computer Architecture, New York, ACM, 1973, pp. 43—51.
29. Nissen S. M., Wallach S. J., An APL Microprogramming Structure, Record of the Sixth Annual Workshop on Microprogramming, New York, ACM, 1973, pp. 50—57.
30. Giloi W. K., Berg H., STARLET — A Computer Concept Based on Ordered Sets as Primitive Data Types, Proceedings of the Second Annual Symposium on Computer Architecture, New York, IEEE, 1975, pp. 201—206.
31. Giloi W. K., Berg H. K., Data Structure Architectures — A Major Operational Principle, Proceedings of the Fifth Annual Symposium on Computer Architecture, New York, ACM, 1978, 175—181.
32. Reddi S. S., Feustel E. A., A Restructurable Computer System, *IEEE Transactions on Computers*, C-27(1), 1—20 (1978).
33. Harrison M. C., A Language-Oriented Instruction Set for the BALM Language, Proceedings of the ACM SIGPLAN-SIGMICRO Interface Meeting, New York, ACM, 1973, pp. 161—168.
34. Tafuelin S., Wikstrom A., Aspects of Compact Programs and Directly Executed Languages, *BIT*, 15(2), 203—214 (1975).
35. Shapiro M. D., A SNOBOL Machine: A Higher-Level Language Processor in a Conventional Hardware Framework, Digest of the Sixth Annual IEEE Computer Society International Conference, New York, IEEE, 1972, pp. 41—44.
36. Snyder A., A Machine Architecture to Support an Object-Oriented Language, TR-209, Laboratory for Computer Science, MIT, Cambridge, MA, 1979.
37. Myamlin A. N., Smirnov V. K., Computer with Stack Memory, Proceedings of the IFIP Congress, 1968, Amsterdam, North-Holland, 1969, pp. 818—823.
38. Feustel E. A., The Rice Research Computer — A Tagged Architecture, Proceedings of the 1972 Spring Joint Computer Conference, Montvale, NJ, AFIPS, 1972, pp. 369—377.
39. Iliffe J. K., Basic Machine Principles, 2nd ed., London, Mcdonald, 1972.
40. Kancler H. C., Architecture of Aerospace Computer Simplifies Programming, *Computer Design*, 15(5), 159—166 (1976).
41. Carpino A., An Architecture Development Tool for an Intermediate Language Machine (ILM), Digest of Papers, Fall Compcon 75, New York, IEEE, 1975, pp. 265—267.
42. Dorocak J. P., Structured Control Operators Implemented on an Intermediate Language Machine (ILM), Digest of Papers, Fall Compcon 75, New York, IEEE, 1975, pp. 268—271.
43. Battarel G. J., Chevance R. J., Design of a High Level Language Machine, *Computer Architecture News*, 6(9), 5—17 (1978).
44. Steel R., Another General Purpose Computer Architecture, *Computer Architecture News*, 5(8), 5—11 (1977).
45. Myers G. J., SWARD — A Software-Oriented Architecture, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland College Park, MD, 1980, pp. 163—168.
46. Broadbent J. K., Coulouris G. F., MEMBERS — A Microprogrammed Experimental Machine with a Basic Executive for Real-Time Systems, Proceedings of the ACM SIGPLAN-SIGMICRO Interface Meeting, New York, ACM, 1973, pp. 154—159.
47. Nielsen W. C., Design of an Aerospace Computer for Direct HOL Execution, Proceedings of the ACM-IEEE Symposium on High-Level Language Computer Architecture, New York, ACM, 1973, pp. 34—42.

48. Melbourne A. J., Pugmire J. M., A Small Computer for the Direct Processing of FORTRAN Statements, *The Computer Journal*, 8(1), 24—27 (1965).
49. de la Guardia M. F., Field J. A., A High Level Language Oriented Multi-processor, Proceedings of the 1976 International Conference on Parallel Processing, New York, IEEE, 1976, pp. 256—262.
50. Balzer R. M., An Overview of the ISPL Computer, *Communications of the ACM*, 16(2), 117—122 (1973).
51. Steele G. L., Jr., Sussman G. J., Design of LISP-Based Processors or, SCHEME, A Dielectric LISP or, Finite Memories Considered Harmful or, LAMBDA, The Ultimate Opcode, AIM-514, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1979.
52. Abrams P. S., An APL Machine, Ph. D. thesis, Stanford University, Stanford, CA, 1970.
53. Hassitt A., Lageschulte J. W., Lyon L. E., Implementation of a High Level Language Machine, *Communications of the ACM*, 16(4), 199—212 (1973).
54. Hassitt A., Lyon L. E., An APL Emulator on System/370, *IBM Systems Journal*, 15(4), 358—378 (1976).
55. Reigel E. W., Faber U., Fisher D. A., The Interpreter — A Microprogrammable Building Block System, Proceedings of the 1972 Spring Joint Computer Conference, Montvale, NJ, AFIPS, 1972, pp. 705—723.
56. Schroeder S. C., Vaughn L. E., A High Order Language Optimal Execution Processor, Proceedings of the ACM-IEEE Symposium on High-Level Language Computer Architecture, New York, ACM, 1973, pp. 109—116.
57. Zaks R., Steingart D., Moore J., A Firmware APL Time-Sharing System, Proceedings of the 1971 Spring Joint Computer Conference, Montvale, NJ, AFIPS, 1971, pp. 179—190.
58. Robinet B. J., Architectural Design of an APL Processor, in Y. Chu, Ed., High-Level Language Computer Architecture, New York, Academic, 1975, pp. 243—268.
59. Bashkow T. R., Sasson A., Kronfeld A., System Design of a FORTRAN Machine, *IEEE Transactions on Electronic Computers*, EC-16(4), 485—499 (1967).
60. Weber H., A Microprogrammed Implementation of EULER on IBM System/360 Model 30, *Communications of the ACM*, 10(9), 549—558 (1967).
61. Burkle H. J., Frick A., Schlier C., High Level Language Oriented Hardware and the Past-von Neumann Era, Proceedings of the Fifth Annual Symposium on Computer Architecture, New York, ACM, 1978, pp. 60—65.
62. Wells M., Denson A., Direct Execution of Programming Languages, *The Computer Journal*, 17(2), 130—134 (1974).
63. Haynes L. S., The Architecture of an ALGOL 60 Computer Implemented with Distributed Processing, Proceedings of the Fourth Annual Symposium on Computer Architecture, New York, IEEE, 1977, pp. 95—104.
64. See the SYMBOL references at the end of Chapter 8.
65. Thurber K. J., Myna J. W., System Design of a Cellular APL Computer, *IEEE Transactions on Computers*, C-19(4), 291—303 (1970).
66. Anderson J. P., A Computer for Direct Execution of Algorithmic Languages, Proceedings of the 1961 Eastern Joint Computer Conference, New York, Macmillan, 1961, pp. 184—193.
67. Bloom H. M., Conceptual Design of a Direct High-Level Language Processor, in Y. Chu, Ed., High-Level Language Computer Architecture, New York, Academic, 1975, pp. 187—242.
68. Haynes L. S., The Architecture of an Algol 60 Computer Implemented with Distributed Processors, Proceedings of the Fourth Annual Symposium on Computer Architecture, New York, ACM, 1977, pp. 95—104.
69. Meggitt J. E., A Character Computer for High-Level Language Interpretation, *IBM Systems Journal*, 3(1), 68—78 (1964).
70. Shaw J. C., Newell A., Simon H. A., Ellis T. O., A Command Structure for

- Complex Information Processing, Proceedings of the 1958 Western Joint Computer Conference, New York, AIEE, 1958, pp. 119—128.
71. Williams R., A Multiprocessing System for the Direct Execution of LISP, Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, New York, ACM, 1978, pp. 35—41.
 72. Thorelli L., Bage G., Description of the High-Level Machine Language of LAX2, TRITA-CS-7901, Royal Institute of Technology, Stockholm, Sweden, 1979.
 73. Mullery A. P., Schauer R. F., Rice R., ADAM — A Problem-Oriented Symbol Processor, Proceedings of the 1963 Spring Joint Computer Conference, Baltimore, Spartan, 1963, pp. 367—380.
 74. Vahey M., Mosteller G., High Level Language Oriented Aerospace Computer, Proceedings of NAECON 1979, New York, IEEE, 1979, pp. 684—690.
 75. Petit J. et al., A Microprogrammed Strategy for HLL Interpretation, *SIG-MICRO Newsletter*, 7(4), 46—68 (1976).
 76. Chu Y., An LSI Modular Direct-Execution Computer Organization, *Computer*, 11(7), 69—76 (1978).
 77. Earnest E. D., Twenty Years of Burroughs High-Level Language Machines, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 64—71.
 78. Yamamoto M. et al., A Cobol Machine Design and Evaluation, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 212—219.
 79. Harris N. R., A Directly Executable Language Suitable for a Bit Slice Microprocessor Implementation, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 40—43.
 80. Yamamoto M., A Survey of High-Level Language Machines in Japan, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 72—79.
 81. Schoellkopf J., PASC-HLL, A High-Level Language Computer Architecture for Pascal, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 222—230.
 82. Bage G., Thorelli L., Partial Evaluation of a High-Level Architecture, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 44—52.
 83. Durrieu G. et al., High-Level Architecture for a Real Time Language LTR, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 130—139.
 84. Ditzel D. R., Patterson D. A., Retrospective on High-Level Language Computer Architecture, Proceedings of the Seventh Annual Conference on Computer Architecture, New York, ACM, 1980, pp. 97—104.
 85. Chen Y. N., Chen K. L., Huang K. C., Direct-Execution High-Level Language Fortran Computer, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 9—16.
 86. Wang P., Liu M. T., The Architecture of a Parallel Execution High-Level Language Computer, Proceedings of the International Workshop on High-Level Language Computer Architecture, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 1—8.
 87. Chu Y., A Jovial Direct Execution Computer, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 17—32.

ГЛАВА 4

НАБОР СРЕДСТВ ДЛЯ СОВЕРШЕНСТВОВАНИЯ АРХИТЕКТУРЫ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

На основе анализа проблем машины традиционной архитектуры может быть разработан набор решений по сокращению семантических разрывов различного вида, рассмотренных в гл. 2. Эти решения не сводятся к таким упрощенным рекомендациям, как увеличение «мощности» системы машинных команд (например, за счет добавления в традиционную архитектуру специального оператора DO, управляющего выполнением цикла команд). Скорее они предполагают более радикальные изменения, например, введение новых принципов организации памяти машины, предопределяя тем самым отход от классической модели машины фон Неймана. Подобные радикальные изменения приводят и к увеличению «мощности» (функциональных возможностей) системы машинных команд, но это — лишь вторичный эффект изменения структуры памяти.

В этой главе указанные изменения рассматриваются с общих позиций, без детализации и конкретного описания или практической реализации. Но поскольку архитектура системы SWARD (детально обсуждаемая в части V) базируется на большинстве из рассматриваемых здесь принципов, ссылки на нее и примеры реализованных в ней решений составляют основу иллюстративного материала данной главы. Для получения более полного представления об основных положениях этой главы читателю следует обратиться к архитектуре вычислительных систем, рассматриваемых в последующих частях книги.

САМООПРЕДЕЛЯЕМЫЕ ДАННЫЕ

Значительным шагом на пути сокращения семантического разрыва между языком программирования и архитектурой машины и существенным отходом от классической модели машины фон Неймана является хранение информации в форме *самоопределяемых* данных; применительно к памяти этот принцип получил

наименование принципа *теговой памяти*. Традиционная архитектура фон Неймана базируется на том, что параметры содержимого памяти носят субъективный характер, существуя лишь в представлении пользователя памятью, т. е. программы. Машина получает информацию о типе хранимых в памяти данных только из потока обрабатывающих команд. Следовательно, предполагается наличие команд, определяющих атрибуты операндов. Так, в Системе 370 проводится четкое различие между командами, выполняющими сложение 32-битовых двоичных

Тег	Содержимое слова памяти
-----	-------------------------

Рис. 4.1. Самоопределяемое слово памяти.

целых чисел, 16-битовых двоичных целых чисел, 32-битовых чисел с плавающей точкой, 64-битовых чисел с плавающей точкой, 128-битовых чисел с плавающей точкой, одиоразрядных десятичных чисел, трехразрядных десятичных чисел и т. д. Альтернативным является такой подход, при котором к ячейке памяти, предназначенной для хранения той или иной информации, добавляется набор битов, описывающих атрибуты (тип данных) содержимого ячейки (рис. 4.1). Термин «ячейка» используется вместо термина «слово», поскольку последнее часто символизирует область памяти определенного размера; однако придерживаться этого ограничения не обязательно.

В простейшем случае (минимальные требования) поле самоопределения данных, или так называемый *тег*, содержит информацию о типе данных рассматриваемой ячейки (т. е. биты тега указывают тип данных, находящихся в ячейке: двоичное целое число, десятичное целое число, число с плавающей точкой, строка символов, адрес и т. п.). Благодаря этому при работе с ячейками можно пользоваться командами, инвариантными к типу обрабатываемых данных. Вместо группы команд ADD (СЛОЖЕНИЕ) машине достаточно иметь одну команду: тип подлежащего выполнению сложения (арифметика целых чисел, арифметика чисел с плавающей точкой и т. п.) машина выявляет путем анализа содержимого тегов операндов каждой команды. Наличие полей тегов «скрыто» от программ, написанных на языках высокого уровня. Теги устанавливаются компиляторами или эквивалентными им программными средствами и используются машиной для определения семантики (назначения) каждой подлежащей выполнению операции, контроля адекватности обрабатываемых данных (например, архитектура машины может предусматривать отказ выполнения сложения адреса с

числом в форме с плавающей точкой) и автоматического преобразования данных (возможен такой вариант построения архитектуры машины, при котором сложение целочисленного операнда с операндом в форме с плавающей точкой допустимо и предполагает предварительное соответствующее преобразование данных).

Развивая принцип использования ячеек памяти с тегом, можно описывать не только тип хранимых в ячейках данных, но и другие их характеристики (в работе [1] предлагается до 32 типов подобных характеристик). Например, при работе с данными переменной длины в теге можно выделить поле для указания длины операнда. Это, в частности, позволило бы избавиться от повторения указателя длины данных в каждой команде, обращающейся к соответствующему слову данных. Так, например, в Системе 370 имеется 15 различных команд ADD. Формат одной из них — ADD DECIMAL — предусматривает наличие двух 4-битовых полей для указания длины обоих операндов команды. С учетом подобных отличий можно сказать, что в Системе 370 имеется 270 различных команд ADD: 14 основных и 256 модификаций команды ADD DECIMAL. В машине с теговой организацией памяти — при условии, что в каждом теге содержится описание типа и длины операнда, — вполне достаточным могло бы оказаться использование только одной команды ADD.

Сам факт принятия решения об использовании теговой памяти и команд, инвариантных к типу и длине обрабатываемых данных, наводит на мысль о других возможных применениях тега. Добавив к нему 1-битовое поле, можно указывать, например, определено ли текущее значение переменной, использующей данную ячейку памяти. А это предоставляет для машины возможность легко выявлять попытки использования неопределенных данных. В микропроцессоре обработки числовых данных 8087 фирмы Intel для этой и других целей используется 2-битовый тег. Тег можно расширить, добавив поле для так называемых *битов захвата*¹⁾. Например, подобный бит может быть определен таким образом, что при обращении программы к ячейке памяти с подобным тегом возникает прерывание, сопровождаемое выполнением определенных процедур отладки.

На рис. 4.2 приведены ячейки памяти четырех типов, использование которых предусмотрено архитектурой системы SWARD (см. часть V). Здесь размер и формат тегов зависят от представляемой ими информации, однако во всех случаях первые четыре бита определяют тип ячейки (точнее, тип дан-

¹⁾ Захват — незапрограммированное прерывание программы при возникновении непредусмотренной ситуации с условным переходом к определенной процедуре. — *Прим. ред.*

ных, для которых ячейка предназначена). На данном рисунке граница между концом тега и началом информационной части ячейки (содержимое ячейки) обозначена стрелкой. Для указанной архитектуры характерна инвариантность (неизменяемость) содержимого тега ячейки в процессе выполнения программы. Исключением являются только ячейки с динамическим определением тегов. Это свидетельствует об ориентации данной архитектуры на языки программирования с детально разработанны-

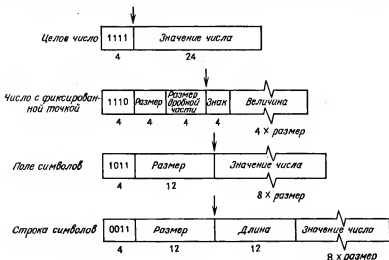


Рис. 4.2. Типы ячеек памяти в архитектуре системы SWARD. (Поле «значение числа» является частным случаем поля «значение символов».)

ми средствами задания типов переменных, т. е. языки, каждая переменная которых имеет строго определенные атрибуты. При этом ячейки не существуют как самостоятельные, независимые элементы памяти; напротив, они принадлежат информационным объектам того или иного типа. Подобный объект может состоять из одной ячейки, набора ячеек, совокупности ячеек и машинных команд или структурно более сложных компонентов.

Вместо того чтобы использовать тег для описания неопределенного значения, последнее описывается как одно из возможных значений содержимого ячеек всех типов. Рассмотрим, например, ячейку для представления числа в форме с фиксированной точкой, а именно десятичного числа в виде целой и дробной частей. Содержимое тега определяет тип ячейки (хранилище числа с фиксированной точкой), количество цифр, записываемых в ячейку в коде BCD, и позицию подразумеваемой

десятичной точки. Положим, что ячейка предназначена для хранения чисел, имеющих следующий формат: XXXX.XX. Если содержимому ячейки в данный момент присваивается значение 5.7, то в памяти оно представляется как E620000570 (шестнадцатеричная форма записи). Если же содержимое этой ячейки не было определено, то имело бы следующий вид: E62FXXXXXX, где X — любой символ.

ДОСТОИНСТВА ТЕГОВОЙ ПАМЯТИ

Расширенные возможности обнаружения ошибок. Совместное хранение данных и их атрибутов позволяет машине обнаруживать в программе ошибки нескольких типов. Например, в архитектуре может быть предусмотрена индикация как ошибочной ситуации, при которой операнды команды оказываются бессмысленными (например, выясняется, что операндом команды умножения является строка символов), несовместимыми (при попытке записи числа с плавающей точкой в качестве адреса) или неопределенными по значению (значение исходного операнда неизвестно). Такой механизм называют защитой типа данных, поскольку он предотвращает попытки выполнения операций над данными, тип которых задан некорректно.

Автоматическое преобразование данных. Теговая организация памяти позволяет машине выполнять автоматическое преобразование данных — операндов команды при условии, что они совместимы, но отличаются длиной или формой представления. Так, в архитектуре машины может быть предусмотрено сложение целых чисел в форме с плавающей точкой; машина автоматически преобразует данные, являющиеся операндами команды ADD. В качестве другого примера можно рассмотреть ситуацию, когда операндами команды являются два числа с фиксированной точкой, находящейся в разных позициях: в процессе выполнения команды машина осуществляет необходимое выравнивание позиций указанных десятичных точек.

Повышенная эффективность выполнения программы. Благодаря теговой организации памяти достигается высокая скорость выполнения команд. Отчасти это является следствием перечисленных выше достоинств теговой памяти. В машине, память которой не использует теги, необходимые преобразования данных выполняют кодовые последовательности, генерируемые компилятором. Скорость выполнения преобразований данных в машине с теговой памятью выше хотя бы потому, что машине, не располагающей подобной памятью, необходимо извлекать из памяти и декодировать команды преобразования данных.

Использование теговой памяти позволяет разработчику ЭВМ создавать более эффективные алгоритмы. Рассмотрим задачу

увеличения на 1 значения 15-разрядного числа, представленного в коде BCD. Большинство машин, использующих подобное представление чисел переменной длины, выполняет такие арифметические операции путем последовательных элементарных приращений, например побайтно в единицу времени. При решении этой задачи вручную — с помощью карандаша и бумаги — естественно прекратить операции при наличии 0 в качестве цифры переноса. Это можно сформулировать следующим образом: прибавить 1 к цифре младшего разряда; если переноса нет, операции прекратить; в противном случае добавить единицу переноса к цифре следующего разряда и т. д. Поскольку Система 370 не использует теговую организацию памяти, результат арифметической операции приходится представлять в соответствующей форме, т. е. в коде BCD. Поскольку машина не может гарантировать, что десятичный операнд, расположенный в памяти, содержит цифры в коде BCD (в той или иной области памяти данной программы байты могут содержать некоторые произвольные данные), операция должна быть выполнена над всеми частями операнда, т. е. над всеми 15 цифрами. Иначе обстоит дело при использовании машины с теговой памятью: согласно принципам, положенным в основу подобной архитектуры, ячейка, предназначенная для десятичных чисел, не может содержать ничего, кроме цифр в коде BCD. Следовательно, защита типа представляемых данных является преимуществом машин с теговой организацией памяти.

Принцип теговой организации памяти позволяет задержать начало процесса «привязки» команды к данным до момента ее выполнения, что, как предполагается, повышает скорость выполнения программы на языке высокого уровня (если правила использования такого языка требуют указанной задержки). Например, при определенных обстоятельствах для программы на языке ПЛ/1 необходимо обеспечить привязку атрибутов данных на этапе выполнения. Это демонстрирует следующий фрагмент программы:

```
ABC: PROCEDURE(C);
DECLARE C CHARACTER(*);
DECLARE D CHARACTER(8) INITIAL('ZYXWVUTS');
IF C=D THEN...
```

Здесь указано, что длина параметра C задается динамически, т. е. принимает размер соответствующего фактического параметра. Поэтому для данного оператора IF компилятор не может сформировать единственную команду сравнения, как этого следовало ожидать в случае, если бы архитектура машины была подобна архитектуре Системы 370. Вместо этого необхо-

димо создание специальных условий, обеспечиваемых программными средствами архитектуры с теговой организацией памяти. Благодаря этому размер фактического параметра передается в процессе вызова процедуры, и тем самым обеспечивается динамическое использование его значения для сравнения переменных *C* и *D*. На пути выполнения этой операции находится еще одно препятствие: перед сравнением длина переменных *C* и *D* должна быть выравнена. Для этого можно произвести усеечение или дополнение *D* пробелами до размера *C*. Чтобы осуществить это, компилятор традиционной машины должен создать некий программный эквивалент подобных возможностей машины с теговой памятью. В случае использования оптимизирующего компилятора языка ПЛ/1, разработанного фирмой IBM, генерируется 49 команд (вместо одной команды машины с теговой памятью). Это же справедливо и в отношении других языков программирования, таких, как Ада.

Меньшее разнообразие типов команд. В отличие от машины с традиционной архитектурой в машине с теговой памятью обычно используются команды, инвариантные к типу обрабатываемых данных, вследствие чего существенно сокращается общий набор команд, их структура приобретает более регулярный характер, отсутствуют многие аномалии, столь типичные для больших наборов команд. В результате архитектура с теговой организацией памяти становится более «пойнтной» компиляторам и программистам.

Архитектуре многих машин присуща корреляция между размером набора команд и числом аномалий в архитектуре. Например, в Системе 370 имеются двоичные данные двух типов: длиной 16 и 32 бит. Набор команд обработки 32-битовых данных включает команды ADD (СЛОЖЕНИЕ), SUBTRACT (ВЫЧИТАНИЕ), MULTIPLY (УМНОЖЕНИЕ) и DIVIDE (ДЕЛЕНИЕ), в то время как в наборе команд обработки 16-битовых данных команда деления отсутствует. Имеется команда HALVE (ДЕЛЕНИЕ ПОПОЛАМ) — деление числа с плавающей точкой на 2. Существование этой команды может вызвать удивление, поскольку имеется соответствующая команда DIVIDE. При этом отсутствует команда DIVIDE BY 2 (ДЕЛЕНИЕ НА 2) для двоичных или десятичных целых чисел (в противоположность распространенному мнению операция арифметического сдвига не эквивалентна делению на 2 [3]).

Более простой компилятор. По сравнению с традиционными машинами для машины с теговой памятью используется более простой и требующий меньшего времени выполнения компилятор. (В гл. 3 этот факт отмечен как положительный побочный эффект, а не основная цель теговой организации памяти.) Процесс генерирования машинных кодов компилятором традицион-

ной машины довольно сложный, поскольку необходим семантический анализ программы для определения того, какие именно команды подлежат генерированию. Например, встретив знак операции $+$, компилятор вынужден анализировать выражения, расположенные по обе стороны от знака, чтобы принять решение о типе команды сложения, подлежащей генерированию. В машине с теговой памятью компилятор просто генерирует команду сложения (принадлежащую набору команд, инвариантных к типу обрабатываемых данных). И если компилятор традиционной машины вынужден осуществлять выбор буквально из сотен возможных кодовых последовательностей для оператора $IF A=B THEN...$, компилятор машины с теговой памятью всегда пользуется одной и той же машинной командой. Простота генерирования машинных кодов в последнем случае объясняется также и тем, что не компилятор, а машина на этапе выполнения осуществляет контроль корректности программы и автоматическое преобразование данных.

Более совершенные средства отладки. Теговая организация памяти вычислительной машины упрощает процесс разработки средств отладки программ. По крайней мере наиболее простые общепользуемые средства, базирующиеся на выдаче дампа памяти, оказываются более эффективными. Это связано с тем, что при теговой организации памяти дмп значительно информативнее для пользователя (благодаря большей наглядности выводимого на печать содержимого областей памяти) по сравнению с традиционным дампом в виде длинных последовательностей двоичных, восьмеричных и шестнадцатеричных цифр. Возможность включения в теги битов захвата делает более вероятной практическую реализацию сравнительно сложных средств отладки. Кроме того, любое архитектурное решение, обеспечивающее сближение интерфейса машины с языком программирования, способствует разработке средств отладки, ориентированных на эти языки.

Независимость программных средств от обрабатываемых данных. Поскольку применение тегов позволяет отложить момент привязки алгоритма работы программы к атрибутам данных на наиболее поздний этап обработки задания (программы и данных), появляется возможность использования инвариантных к типу данных команд для написания программы и реализации принципа независимости программных средств от данных в структуре базы данных. Указанный принцип независимости позволяет прикладным программам обрабатывать отдельную запись базы данных (логическую запись) без учета ее физического представления (физической записи). В простейшем случае, например, прикладная программа может «полагать», что содержимое некоторого поля — десятичное число, хотя в

реальной базе данных в этом поле может находиться двоичный эквивалент этого числа.

Практическая реализация рассматриваемого принципа в существующих вычислительных системах затруднена тем, что их машинные команды содержат информацию о типе и длине своих операндов, а поэтому при внесении изменений в определение базы данных требуется повторное компилирование ее программных средств. Что же касается машин с теговой организацией памяти, то благодаря использованию команд, инвариантных к типу данных, и при условии распространения теговой организации на внешнюю память, практическое осуществление принципа независимости программных средств от данных становится реальностью.

Уменьшение необходимого объема памяти. Речь идет о достоинстве машин с теговой организацией памяти, которое по интуитивным, но ошибочным соображениям многим кажется спорным и даже противоречивым. В таких машинах для размещения программы и данных требуется меньший объем памяти, чем в машинах с традиционной, «нетеговой» организацией памяти. Объяснению этого явления посвящен следующий раздел.

ПАМЯТЬ, НЕОБХОДИМАЯ ДЛЯ РАЗМЕЩЕНИЯ ТЕГОВ

Получившая распространение критика принципа теговой организации памяти обычно основана на интуитивном предположении, что стоимость системы, использующей теги, возрастает вследствие увеличения требуемого объема памяти. Утверждают, например, что добавление 4-битового тега к 32-битовым словам машины увеличивает стоимость ее памяти на 12,5%. Однако в данном случае интуиция явно подводит: машины с теговой памятью требуются меньшее количество битов памяти, чем машины с традиционной архитектурой.

Прежде всего следует отметить, что традиционным машинам присуща значительная избыточность хранящейся в памяти информации об атрибутах данных. Так, если число с плавающей точкой адресуется многими командами, имеет место избыточность в повторной идентификации каждой такой командой данного числа как операнда в форме с плавающей точкой (в поле кода операции команды имеются биты, позволяющие отличить операцию арифметики чисел с плавающей точкой от арифметики чисел других типов). Поскольку к переменным адресуются, как правило, несколько команд, представляется рациональным для сведения расхода памяти к минимальному устранять избыточность в повторяемых командах за счет хранения информации о типе операнда команды вместе с самим значением операнда

(хранение информации в статическом виде).

В качестве простого примера рассмотрим машину X со 150 типами различных команд, код операции каждой из которых занимает поле длиной 8 бит. Альтернативой этой машины будем считать машину Y с теговой памятью, каждая ячейка которой содержит 3-битовый тег. Команды машины Y являются инвариантными к типу обрабатываемых данных. Это позволяет предположить, что для решения тех же задач, которые стоят перед машиной X, машине Y окажется достаточно только 50 типов команд, код операции каждой из которых может быть описан 6 бит.

Для сравнения объема памяти, расходуемой той и другой машинами, следует определить (в битах) количество кодов операции и тегов, необходимых некоторой программе, — назовем эту сумму В. При этом предполагается, что все остальные потребности в памяти являются неизменными для обеих машин. Если I есть количество машинных команд программы, то для машины X справедливо равенство $B = 8 \cdot I$. Для машины Y имеет место соотношение $B = 6 \cdot I + 3 \cdot P$, где P — число операндов в программе. В предположении, что каждая команда обеих машин манипулирует двумя операндами и к каждому операнду в программе производится в среднем R обращений, для машины Y получим равенство $B = 6 \cdot I + 6 \cdot I/R$. Отношение величины В машины Y к величине В машины X равно $0,75 \cdot (1 + 1/R)$. Если значение этого отношения меньше 1, то машина с теговой памятью расходует меньше памяти.

Для количественной оценки полученного отношения необходимо знать, чему равно значение параметра R, т. е. значение среднего числа обращений к операнду одной команды. Естественно ожидать, что его величина больше 1. Критическим является значение R, равное 3; при этом условии для обеих машин требуется одинаковый объем памяти. Параметр R не относится к числу часто измеряемых показателей работы программы. Согласно результатам экспериментов с одним набором программ [4], его величина равна 10,4. При оценке среднего числа обращений к операнду в программах на языке КОБОЛ эта величина составила 3,5 [5]. Если воспользоваться значением 10,4 как типичной величиной для программы в машинных кодах, то для размещения кодов операции команд и тегов данных в машине с теговой организацией памяти потребуется 82% объема памяти, необходимой для выполнения тех же программ в машине с традиционной архитектурой. Если же положить R равным 3,5, то указанная величина станет равной 96%.

Итак, для машины с теговой организацией памяти требуется меньший объем памяти благодаря устранению избыточности информации в кодах операции команд. Этот противоречащий

интуиции феномен подтверждает одно из положений, сформулированных в гл. 1: только глубокое понимание языков программирования и специфических характеристик программ позволяет разработчику архитектуры вычислительной системы принимать рациональные решения.

Таблица 4.1. Ресурсы машины, расходуемые на автоматическое преобразование данных и проверку результата на переполнение

Оператор	Машинные команды	Размер, байт	Время выполнения, мкс
Без контроля на переполнение $D=D+A$; $D=D+I$;	1 13	6 64	13,1 407,6
С контролем на переполнение $D=D+A$; $D=D+I$;	7 19	38 92	57,7 448,1

Использование тегового принципа организации памяти позволяет уменьшить ее объем и за счет устранения избыточности другого типа: команд, повторно генерируемых компилятором для выполнения функций контроля и преобразования данных на этапе выполнения программы. Положим, что в программе на языке ПЛ/1 переменная I объявлена как FIXED BINARY(31), а переменные A и D — как FIXED DECIMAL(4,2). В Системе 370 при использовании оптимизирующего компилятора языка ПЛ/1 машинный код, генерируемый для оператора $D=D+A$, занимает 6 байт памяти, а для размещения кода для оператора $D=D+I$ требуется 64 байт (табл. 4.1), т. е. происходит увеличение на 967%. Это объясняется тем, что во втором случае компилятор генерирует код для преобразования данных. Если задать необязательный параметр проверки операции записи в область D на переполнение, то для оператора $D=D+I$ генерируется 92 байт объектного кода. Согласно табл. 4.1, увеличение времени выполнения команд еще значительнее: выполнение оператора $D=D+I$, содержащего операнды разных типов, составляет 3011% времени выполнения оператора $D=D+A$ (измерения проведены на ЭВМ модели 145 Системы 370).

Указанные операции контроля и преобразования данных представляются обычно в виде машинных кодов, встроенных в том месте объектного кода программы, где они необходимы. Однако такое решение не является обязательным. Коды этих операций можно формировать в виде подпрограмм, вызываемых по мере необходимости. Единственная причина, по которой

так обычно не поступают, заключается в чрезвычайной неэффективности реализации такого варианта. Теговая организация памяти позволяет устранить упомянутую выше избыточность, возложив выполнение указанных функций на машину. Конечно, эти функции необходимо реализовать в какой-то определенной форме, например в виде информации, занимающей некоторый объем управляющей памяти машины с микропрограммным управлением. Однако в машине с теговой памятью эти функции физически реализуются только один раз (в заданной области управляющей памяти), вместо того чтобы дублироваться сотни или тысячи раз в объектном коде программы, размещаемой в основной памяти машины.

Ранее при рассмотрении влияния тегов на скорость выполнения программы отмечалось, что при использовании некоторых языков программирования требуется динамическая привязка (назначение) атрибутов данных командам. При использовании памяти без тегов это условие реализуется с помощью машинных кодов, генерируемых компилятором, что, безусловно, влечет за собой дополнительный расход памяти. В рассмотренном выше примере (для переменных C и D) решение подобной задачи компилятором сопровождается генерированием 156 байт машинного кода. Если же динамической привязки атрибутов данных командам не требуется, например, если C определена как переменная фиксированной длины, то оказывается достаточно 6 байт подобного кода.

Приведем последний довод в пользу утверждения, состоящего в том, что тегам требуется не так много памяти, как кажется на первый взгляд. Тег не является физически неотъемлемой частью каждого элемента данных теговой памяти. Например, нет необходимости хранить тег для каждого элемента массива. Поскольку, согласно определению массива, его элементы имеют идентичные атрибуты, можно ограничиться одним тегом для всего массива. По той же причине не требуется наличия тега для каждого элемента строки символов; достаточно иметь один тег, определяющий атрибуты всех элементов. Более подробно этот вопрос анализируется в следующем разделе.

НЕДОСТАТКИ ТЕГОВОЙ ПАМЯТИ

Как и следовало ожидать, теговой памяти присущи недостатки. Одним из них является сам факт строгого определения типа данных, что иногда трудно совместимо с возможностями существующих языков программирования, не обладающих адекватной строгостью и точностью определения типов переменных (см. упр. 4.1). Другим недостатком теговой памяти являются ограничения, накладываемые ею на скорость выполнения операций. И это наряду с отмеченными выше достоинствами те-

говой памяти при сравнении ее быстродействия с быстродействием традиционной памяти машины.

Указанные ограничения вызваны тем, что при использовании теговой памяти процесс привязки атрибутов данных командам откладывается на этап выполнения программы, когда эта процедура производится многократно (с каждым выполнением команды), а не однократно, как это осуществляется на более ранней стадии (например, на этапе компилирования). Иначе говоря, эту проблему можно сформулировать следующим образом: если в программе на языке высокого уровня указывается сложение двух целочисленных переменных, почему бы на этапе компилирования не сгенерировать команду целочисленного сложения, вместо того чтобы сначала создавать машинную команду сложения, инвариантную к типу операндов, а затем на этапе выполнения извлекать содержимое тегов, проверять соответствие фактических значений операндов их описанию как числовых данных и использовать эти данные для принятия решения о выполнении целочисленного сложения?

Для ответа на этот вопрос необходимо взвесить все «за» и «против» теговой памяти, а именно: 1) отмеченные здесь отрицательные и рассмотренные выше положительные факторы, влияющие на скорость выполнения программ; 2) чрезвычайную неэффективность привязки команд к типам операндов на этапе выполнения при использовании машин с памятью без тегов и наличие в языках программирования конструкций, делающих такую привязку желательной; 3) крайне незначительное влияние возможных изменений скорости выполнения программ на производительность системы в целом при существенном проявлении нескольких очевидных достоинств теговой организации памяти.

Принцип теговой организации памяти нельзя считать новым, однако вопрос о сравнении скоростей выполнения программ традиционными машинами и машинами с теговой памятью до сих пор не нашел достаточного освещения в литературе. Можно только предположить, что при выполнении программ, не содержащих ошибок, с небольшим числом необходимых преобразований данных и фиксируемой на этапе компилирования семантикой выполнения команд машина с теговой памятью работает медленнее, чем традиционная машина. Во всех остальных случаях быстродействие машины с теговой памятью выше.

ТЕГИ И ДЕСКРИПТОРЫ

Машина, в которой используются дескрипторы данных, занимает промежуточное положение между машиной с теговой организацией памяти и традиционной машиной. *Дескриптор* — это

информация о параметрах (атрибутах) данных, играющая роль косвенного адреса ячейки памяти с данными. Сравнение принципов использования тегов и дескрипторов демонстрирует рис. 4.3. В машине, использующей дескрипторы, команды содержат ссылки на дескрипторы, несущие информацию об атрибутах данных, которые в свою очередь обычно содержат ссылки на адреса областей памяти, хранящих значения операндов.

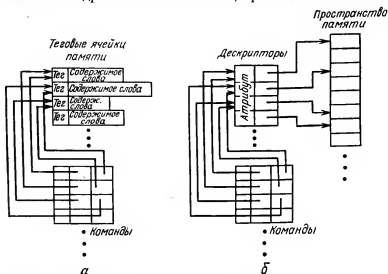


рис. 4.3. Теги (а) и дескрипторы (б) во взаимосвязи командами и памятью.

дов команд. Основные различия между принципами использования тегов (рис. 4.3, а) и дескрипторов (рис. 4.3, б) заключаются в следующем:

1. Дескрипторы предполагают наличие дополнительного уровня адресации, а следовательно, дополнительного времени для предварительной обработки соответствующих адресов и дополнительного места в памяти.

2. Дескрипторы принято рассматривать как часть программы, а не часть данных. Если полагать, что программа и данные — это отдельные информационные единицы (как это и имеет место при использовании в прикладных программах разделяемых данных, файлов, баз данных и т. п.), то нельзя считать дескрипторы четко определенными разграничителями типов этих единиц. Например, очередная прикладная программа может описывать пространство данных другими дескрипторами, определяющими значения, недействительные согласно определениям дескрипторов предыдущей программы.

3. Согласно п. 2, дескрипторы не позволяют достичь такой независимости описания данных, которая возможна при использовании тегов.

4. Согласно п. 2, дескрипторы играют роль некой маски, посредством которой можно «рассматривать» содержимое памяти, в то время как теги содержат описание хранимой в памяти информации.

5. Дескрипторы выполняют индексирование линейного пространства памяти, т. е. не позволяют отказаться от традиционного представления памяти (попыткам такого отказа, в частности, и посвящена данная глава).

На основании перечисленного дескрипторы можно рассматривать как определенный, не лишенный недостатков шаг в направлении создания теговой памяти. В архитектуре большинства машин, описываемых в данной главе, имеются теги или дескрипторы, а в некоторых случаях (например, в ЭВМ В6700 фирмы Burroughs) и то и другое. На внешнем уровне архитектуры Системы 38 фирмы IBM используются не теги, а дескрипторы.

НАБОРЫ САМООПРЕДЕЛЯЕМЫХ ДАННЫХ

Понятие «самоопределяемые данные» можно расширить, если включить в него не только одиночные данные того же вида, но и многомерные совокупности этих данных, такие, как массивы,

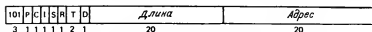


Рис. 4.4. Дескриптор слова ЭВМ В6700 фирмы Burroughs.

таблицы, структуры или записи. Хотя для их описания возможно использование тегов, чаще всего для этих целей применяют указанные выше дескрипторы. В качестве примера использования последних рассмотрим описание массивов в вычислительной системе В6700, имеющей длину слова, равную 51 бит, первые три бита которого — тег. Если содержимое тега равно 101, то данное слово — дескриптор. На рис. 4.4 показан дескриптор одного из возможных типов — дескриптор слова. Бит Р указывает, находятся данные в основной памяти или во внешней. Бит С предназначен для указания, является ли данный дескриптор копией другого дескриптора. Бит I используется для указания, описывает ли данный дескриптор всю совокупность данных или лишь один элемент этой совокупности. Бит S указывает, занимают ли описываемые данные непрерывную область памяти или же расположены в виде сегментов в ее раз-

ных частях. Единичное значение бита R означает, что описываемые данные можно только читать (копировать). Если в рассматриваемом случае биты T равны 0, то данный дескриптор описывает слово, а не строку. Бит D указывает на точность (одинарную или двойную) представления описываемых данных. Если бит I равен нулю, то следующее за ним поле дескриптора определяет число элементов, описываемых дескриптором, а по-

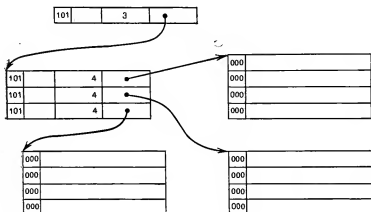


Рис. 4.5. Представление двумерного массива.

следнее поле содержит ссылку на начало описываемых данных.

В системе B6700 дескрипторы могут объединяться в древовидные структуры для описания многомерных объектов памяти. На рис. 4.5 показана подобная структура в виде массива 3×4 слов. Дескриптор массива содержит ссылку на 3-элементный вектор дескрипторов, который в свою очередь указывает на 4-элементные векторы слов данных. Одним из следствий такой организации является то, что машина ответственна за вычисление индексов элементов массивов и контроль принадлежности индексов диапазону возможных значений. Для обращения к конкретному элементу в программе достаточно указать имя дескриптора массива и значения индексов элемента. Машина вычисляет адрес нужного элемента и в то же самое время проверяет, не выходят ли индексы за границы допустимых значений.

В качестве другого примера использования дескрипторов для описания массивов может служить вычислительная система MU5 [6] с дескрипторами двух типов — вектора и строки. Дескриптор вектора состоит из трех полей длиной 8, 24 и 32 бит

соответственно. Содержимое первого поля определяет длину каждого элемента вектора (1, 2, 4, 8, 16, 32, 64 или 128 бит), содержимое второго поля — размер вектора (верхнюю границу числа элементов), содержимое третьего поля — начало области памяти, в которой размещается вектор. Как и в системе B6700, здесь элементы вектора сами могут быть дескрипторами.

Наиболее распространенным видом массива, особенно в задачах нечисленного анализа и делопроизводства, является таблица данных. Обычно под этим термином понимают одномерный массив упорядоченного набора данных, возможно, разнородных по типу. Например, определение 10-элементного массива TAB (каждый элемент которого состоит из 8-символьного и целочисленного полей) в программе на языках ПЛ/1 и Ада имеет следующий вид:

DECLARE 1 TAB(10),	type PERSON is
2 NAME CHAR(8),	record
2 AGE FIXED BINARY;	NAME : STRING(1..8);
	AGE : INTEGER;
	end record;
	TAB : array(1..10) of PERSON;

По внешнему виду имена NAME и AGE фрагмента программы на языке ПЛ/1 ничем не отличаются от имени массивов. При использовании же имени NAME в выражении TAB.NAME(1) в поле NAME размещается значение 1-го элемента массива TAB.

В системе MU5 наряду с дескрипторами используются подобные им по своим функциям *фиктивные векторы* (dope vector) для описания массивов с нижним пределом, меньшим 1, и столбцов таблиц, именуемых *срезами* (slices). Фиктивный вектор состоит из трех 32-битовых полей. Первые два поля содержат значения нижнего и верхнего пределов, а третье поле — значение «шага» — сдвига среза в пределах элемента массива. В приведенных выше фрагментах программ фиктивные векторы формируются для имен NAME и AGE.

Несмотря на существенное сближение принципов организации памяти систем B6700, MU5 и им подобных с соответствующими элементами структуры языков программирования, возможны и дальнейшие шаги в этом направлении. Например, дескрипторы делают «видимой» структуру памяти, предоставляемой элементам массива. От использования дескрипторов можно отказаться и обращаться непосредственно к занимаемому массивом области памяти, нарушая тем самым принципы организации памяти. Стремление к тому, чтобы структура памяти, занимаемой массивом, была «видна», накладывает определенные ограничения: элементы массива должны располагаться в смежных областях памяти, все строки и все столбцы должны

быть одинаковой длины и т. п. К таким ограничениям относится и отказ от использования одного (общего) тега для всех элементов массива (см. предыдущий раздел). Например, в системе B6700 каждый элемент массива имеет тег, который, следовательно, несет избыточную информацию.

Стремясь устранить эти недостатки, разработчики системы SWARD (часть V) предложили описание массива как данных особого типа, отказавшись тем самым от использования дескрипторов. Формат ячейки (области) памяти, предоставляемой

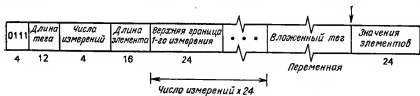


Рис. 4.6. Ячейка памяти для массива в машине системы SWARD.

массиву, показан на рис. 4.6. Особый интерес представляет назначение двух последних полей.

Поле вложенного тега — это последнее из полей, которые предоставлялись бы тегам, если для устранения избыточности они не были бы заменены одним этим тегом согласно подходу, рассмотренному в предыдущем разделе. Вместо хранения тега с каждым элементом массива атрибуты элементов, т. е. содержимое тегов элементов, определяются в теге массива как часть его содержимого; содержимое этого тега — скалярная величина.

Последнее поле, изображенное на рис. 4.6, предназначено для хранения значений элементов массива и имеет постоянную фиксированную длину независимо от размера (числа элементов) массива. Согласно принципам, заложенным в архитектуре системы SWARD, в этом поле должны размещаться все элементы массива. Однако программе содержимое этого поля «не видно», поэтому создание средств для его формирования вменяется в обязанность системным программистам — лицам, ответственным за выбор конкретной конфигурации системы и ее воплощение. В таком случае естественно предположение о целесообразности использования указанного поля не для размещения элементов массива, а для хранения ссылок на адреса их местоположения, что обеспечивало бы косвенную адресацию к этим элементам. Однако это означает определенную маскировку действительного местонахождения элементов массива в памяти.

Помимо ячейки памяти для массива, именуемой в системе SWARD ячейкой «массив», архитектура SWARD предусматривает использование ячейки памяти другого типа, называемой ячейкой «запись» и предназначенной для описания структур (записей) и их массивов. Более подробно об этом идет речь в части V данной книги.

ДОСТОИНСТВА НАБОРОВ САМООПРЕДЕЛЯЕМЫХ ДАННЫХ

Как и следовало ожидать, распространение принципа самоопределения данных на такие информационные объекты, как массивы и структуры, обеспечивает вычислительной системе ряд преимуществ, подобных тем, которые проявляются при использовании теговой памяти.

Расширенные возможности обнаружения ошибок. Машинна с самоопределяемыми данными способна обнаруживать такие распространенные ошибки в программе, как использование элементов массивов с индексами, выходящими за границы допустимых значений. Затраты машинного времени на соответствующую проверку можно считать пренебрежимо малыми по сравнению с рассматриваемым ниже выигрышем в скорости выполнения программ. В гл. 2 показано, почему непрактична реализация подобной проверки программными средствами: при использовании в программе на языке ПЛ/1 необязательного параметра SUBSCRIPTRANGE для организации указанной проверки оператора

$$C(I,J) = A(I,J) + B(J,I);$$

Система 370 выполняет 57 команд (вместо 17 без проверки); при этом длина машинного кода равна 274 байт (вместо 62).

Повышенная эффективность выполнения программы. Поскольку машинна выполняет индексирование массивов непосредственно, а не с помощью предназначенного для этих целей машинного кода, генерируемого компилятором, поиск элементов массива осуществляется быстрее, чем в случае использования машин с традиционной архитектурой. Это связано главным образом с сокращением числа обращений к памяти за командами и с уменьшением числа операций по декодированию команд. Если принцип инвариантности команд к типу обрабатываемых данных распространяется на такие данные, как целые массивы, то операции над массивами становятся более эффективными по тем же причинам, по которым более эффективны команды, инвариантные к ранее рассмотренным типам данных. Например, всем элементам массива можно присвоить определенное значение, выполнив однократно одну операцию над всем массивом

вместо многократного повторения в цикле этой операции над каждым элементом.

Уменьшение необходимого объема памяти. Благодаря перечисленным выше достоинствам машины с самоопределяемыми данными генерируют меньшее количество машинных команд, что в свою очередь сокращает объем памяти, необходимый для размещения программы. Приведенный выше оператор языка ПЛ/1 формирует в Системе 370 семнадцать машинных команд, занимающих 62 байт памяти. Большинство этих команд производится машиной с традиционной архитектурой с целью выполнения операций индексирования массива; эти команды генерируются для каждого обращения к массиву, создавая избыточную информацию в памяти.

Расширенные возможности реализации системы. Вводя изменения в архитектуру системы, разработчик вправе ожидать, что использование дополнительных параллельных алгоритмов работы системы должно повысить результирующую скорость выполнения программ. Примером может служить индексирование элементов массива. Если, предположим, в смежных областях памяти размещается двумерный массив, то обращение к его элементу с индексами I и J вызывает обычно вычисление по

Адрес элемента = Базовый адрес массива + I \cdot N — N + J — 1.

Если индексирование элементов массива выполняется непосредственно машиной, то возможно совмещение операции умножения с операциями сложения и вычитания. Если же в машине предусмотрены команды, инвариантные к типу данных и ориентированные на операции с целыми массивами, то появляется возможность параллельной обработки нескольких элементов массива. Процессор, располагающий кэш-памятью, может заблаговременно загрузить буфер необходимой информацией, вместо того чтобы запрашивать значение базового адреса при выполнении операций над массивом.

ТЕГИ, ОБЪЕКТЫ И ДЕСКРИПТОРЫ

В предыдущем и данном разделах демонстрируется разница между идентификацией данных посредством тегов и с помощью дескрипторов. Введем сопутствующее им понятие «объект». О семантическом различии между этими тремя понятиями свидетельствуют следующие обстоятельства: в системе B6700 фирмы Burroughs тег сопровождает каждое физическое слово памяти, а дескрипторы используются для описания векторов; архитектура машины SWORD основана на применении тегов и объектов; система iAPX 432 фирмы Intel предполагает исполь-

зование объектов и некоторых типов необязательных к употреблению дескрипторов, но исключает применение тегов.

Объект — это некоторое абстрактное понятие, используемое при построении модели вычислительной системы. Обычно объект принято характеризовать следующим образом:

1. Объект — это совокупность взаимосвязанных элементов информации, которая может включать непосредственно данные, сведения о состоянии системы и др.

2. Элементы или части объекта не могут иметь независимое друг от друга «время жизни», т. е. они создаются и уничтожаются одновременно.

3. К объекту обычно адресуются как к единому целому, а не путем индивидуальной адресации его отдельных частей.

4. Для выполнения преобразований с объектами вычислительная система располагает некоторым набором машинных команд.

5. Внутренняя структура объекта остается «невидимой» (скрытой).

6. Объект содержит идентифицирующую его информацию (информацию самоидентификации), запрещающую программе выполнять операции над объектом другого, отличного от данного типа.

В этих характеристиках объекта программисты могут найти много общего с понятием «абстрактные данные», хотя в рассматриваемом случае возникновение абстрактного понятия «объект» связано с машиной. Например, в рамках систем SWARD и iAPX 432 существует объект, именуемый *порт*. Объект такого типа используется для пересылки данных от одного процесса (задания) к другому. Внутренняя структура объекта «порт» программу «не интересует»; она оперирует им как операндом, адресуясь к нему как к набору машинных команд.

Возможную взаимосвязь данных тегового типа и объектов можно увидеть в принципах, положенных в основу архитектуры SWARD. Здесь объектом может быть содержимое одной ячейки данных тегового типа или набора таких ячеек, а также совокупность какой-либо другой информации, исключающей содержимое ячеек данных. Машина позволяет программе динамически разрешать использование данных определенного типа (аналогично тому, как это делают оператор ALLOCATE языка ПЛ/1 или средства описания типов данных языка Ада для создания массива, строки, записи, одиночного целого числа и т. д.). Такая возможность использования данных называется объектом «данные», поскольку это — одиночная ячейка данных, «время жизни» которой не зависит от других ячеек данных. Объект другого типа — «модуль» — содержит машинные команды, набор ячеек данных тегового типа (для любых статических

или собственных переменных, используемых этими командами) и другую информацию о состоянии системы. Объект «порт» является примером объекта такого типа, который не содержит ячеек данных.

ОБЛАСТИ САНКЦИОНИРОВАННОГО ДОСТУПА

Архитектуре современных машин присущи негибкие и грубые средства защиты данных от несанкционированного доступа; в большинстве микропроцессоров такие средства вообще отсутствуют. Обычно указанные средства используются для защиты данных и программного обеспечения системы (например, операционной системы) от модификации или инспекции прикладными программами, а также одной прикладной программы от несанкционированного доступа со стороны другой прикладной программы. Однако такие средства не обеспечивают защиту, во-первых, программы (процесса) от самой себя или, говоря точнее, одной программной секции (например, процедуры) от другой и, во-вторых, программы от программного обеспечения системы. Поэтому было предложено формирование небольших областей санкционированного доступа — *доменов* — с целью достижения большей детальности расчленения всего адресного пространства системы на зоны, защищенные от несанкционированного доступа. Речь идет как о защите от случайных обращений извне или от программных ошибок, так и от преднамеренных обращений, защита от которых определяется требованиями конфиденциальности хранящейся в машине информации. *Область санкционированного доступа* — это независимое локальное адресное пространство, определяющее совокупность адресов, которые могут использоваться или формироваться некоторым набором команд [7—10]. Небольшая область санкционированного доступа (называемая, как указано выше, доменом) предполагает разбиение всего адресного пространства на некоторое, весьма значительное число более мелких локальных адресных пространств, когда на одну прикладную программу (или процесс) либо программу операционной системы приходится не одна, а несколько таких областей.

Появилась идея предоставления каждой программной секции одного такого домена. Определение понятия *программная секция* зависит от принципа адресации и допустимого набора идентификаторов конкретного языка программирования (при этом архитектура вычислительной системы должна быть достаточно гибкой для адаптации к различным языкам программирования). Как правило, программная секция определяется как независимо компилируемая программная единица (пакет, про-

цедура, подпрограмма, программа-монитор и т. п.). Поэтому вместо размещения программы полностью в одном большом адресном пространстве (т. е. одном подпространстве адресного пространства системы) каждому модулю предоставляется отдельное адресное пространство, доступное только его локальным переменным. А это влечет за собой проблему организации связи между доменами, которая решается с помощью механизма управления подпрограммами (описываемого в сле-

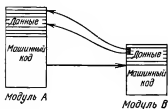


Рис. 4.7. Два домена санкционированного доступа.

дующем разделе), регулирующего процесс обмена информацией между формальными и фактическими параметрами. Передачу фактических параметров подпрограмме, занимающей некоторый домен, можно рассматривать как временное расширение ими этого домена (рис. 4.7). В идеальном случае домен В расширяется только на время передачи ему из домена А фактических параметров (поскольку домен В не имеет адресных связей с какими-либо иными элементами домена А), и это расширение исчезает с возвратом управления от В к А.

Понятие домена часто сопровождается понятием «точка санкционированного входа». В пределах каждого домена программисту может быть предоставлена возможность определения одной или нескольких специфических точек входа, которые исключают возможность вызова модуля, расположенного в домене, путем обращения к любой его команде.

Достоинства использования доменов

Улучшение отладки программ. Разбиение адресного пространства программы на домены, предоставляемые отдельным программным секциям, повышает степень локализации возможных ошибок. В машине, имеющей традиционную архитектуру, ошибка адресации может оказать воздействие на любую часть программы, а ошибка, содержащаяся в одной части операционной системы, может повлиять на работу других ее частей или прикладные программы. В машине с разбиением адресного пространства на домены сфера действия возможной ошибки ограничена пределами домена, в котором она находится (включая лю-

бые расширения данного домена за счет передачи в него фактических параметров). А это повышает вероятность обнаружения ошибки на более ранней стадии выполнения программы и, что также важно, в «точке» ее возникновения.

Повышение надежности защиты программы от несанкционированного доступа. Благодаря доменам информация, принадлежащая одной части программы или процесса, защищена от воздействия других ее частей. Таким образом, обеспечивается защита от несанкционированного доступа, отсутствующая в традиционных машинах. Принцип доступа модуля только к тем данным, которые необходимы для его функционирования, может быть назван *принципом наименьших привилегий*.

В подобных средствах защиты от несанкционированного доступа нуждаются программы, состоящие из секций различного уровня требуемой надежности подобной защиты, или программы, обращающиеся за обслуживанием к другим программным средствам (например, к программам операционной системы или модулям пакета подпрограмм), имеющим другой уровень надежности защиты. Предположим, что модуль А (рис. 4.7) содержит «чувствительные» к несанкционированному доступу данные и нуждается для выполнения какой-то работы в обращении к модулю В. Последний либо принадлежит другому программисту, либо является частью подпрограмм общего назначения, либо включен в набор программ операционной системы. Проблема защиты данных может возникнуть, поскольку существует вероятность того, что модуль В является как бы тройским конем, т. е. чужой программой с «замаскированными намерениями», вводимой внутрь «крепостных стен» модуля А. Если в результате вызова модуля В последний получает доступ ко всем данным модуля А, то, выполняя требуемые от него функции, модуль В может анализировать данные модуля А, нуждающиеся в защите от подобного доступа. Если адресное пространство разделено на домены, возможности адресации модуля В ограничены только фактическими параметрами, передаваемыми ему при вызове.

Механизм защиты от несанкционированного доступа посредством доменов применим и к вызываемым модулям. Например, модуль В может содержать нуждающиеся в защите данные, формируемые в виде некоторой базы данных в периоды вызова модуля В другими модулями, например модулем А. При отсутствии областей, защищенных от несанкционированного доступа, не исключается возможность для модуля А, имеющего адрес точки входа в модуль В, выполнить арифметические операции над этим адресом и получить доступ к «секретным» данным модуля В. Возможна также и следующая ситуация несанкционированного доступа при отказе от использования доме-

нов: модуль А, получив назад управление от вызывавшегося модуля В, просматривает адресное пространство стека подпрограмм с целью анализа значений локальных переменных модуля В.

Принудительное разбиение программ на модули. Известно (например, [11]), что простое разбиение программы на отдельные модули не обязательно приводит к реализации всех преимуществ модульного программирования. Разбиение программы на отдельные модули иногда сопровождается отступлениями от общепринятых правил взаимодействия модулей явным образом через списки параметров. Так, при стремлении добиться хорошо структурированной программы в некоторых случаях допускается прямой доступ из одного модуля к данным другого модуля. Использование механизма защиты от несанкционированного доступа с помощью доменов позволяет исключить указанные нарушения правил обмена информацией между модулями и, следовательно, обеспечить корректность разбиения программы на модули, причем такое разбиение, как правило, бывает неизбежным.

Разбиение адресного пространства машины на домены предъявляет к ее архитектуре ряд требований, не являющихся очевидными на первый взгляд. Например, требуется механизм, гарантирующий вызванному модулю, получившему доступ к фактическому параметру, невозможность доступа к другим данным домена, в котором этот параметр находится. Другим требованием является гарантия входа в модули только через определенные точки входа. Более подробно эти вопросы освещаются в части V (архитектура SWARD), а также в технической литературе, содержащей описание еще одного типа архитектуры с доменами [12], средств передачи фактических параметров [13] и реализации доменов программными средствами в ЭВМ PDP-11 [8].

УПРАВЛЕНИЕ ПОДПРОГРАММАМИ

Важным шагом на пути дальнейшего сокращения семантического разрыва между архитектурой машины и языком программирования является отображение структуры программы в архитектуре машины, причем особое значение имеет соответствие архитектуры средствам организации подпрограмм или процедур, используемым программами. Необходимость в этом обусловлена тем, что указанные средства в традиционных машинах реализуются программным путем, что является весьма продолжительным процессом. Например, при обращении к процедуре в типичной программе на языке высокого уровня соответствующий этой программе объективный код, генерируемый ком-

пилятором, должен выполнить следующие действия: динамически (т. е. в момент вызова) выделить в памяти область, называемую «запись активации» и предназначенную для размещения локальных переменных и информации о состоянии вызываемой процедуры; добавить эту запись к поддерживаемому программными средствами стеку записей активации для данной программы; записать информацию о состоянии вызывающей процедуры на хранение в ее запись активации; присвоить начальные значения содержимому локальной памяти и параметрам вызываемой процедуры; передать управление вызываемой процедуре. Как отмечено в гл. 2, этот процесс выполняется весьма часто и, следовательно, является предметом изучения соотношений между средствами аппаратного и программного обеспечения. Согласно результатам исследований [14, 15], на каждые 50—100 машинных команд имеет место один вызов подпрограммы.

Сокращение указанного семантического разрыва предполагает такую модификацию архитектуры традиционной машины, которая позволила бы реализацию аппаратными средствами некоторых или всех перечисленных функций. Как и в случае модификации других сторон взаимоотношений аппаратных средств и программного обеспечения с целью сокращения семантического разрыва между ними, здесь возможна различная степень уменьшения этого разрыва. Ограничим рассмотрение двумя подходами. При первом подходе существенного изменения традиционной модели памяти машины не требуется, но предполагается введение команд, помогающих реализовать средства организации подпрограмм. При втором подходе указанная модель меняется и память более «похожа» на ее представление в языках программирования. Пользуясь понятием «стек активации», первый подход можно было бы определить как применение *видимого (явного)* стека активации, а второй — применение *скрытого (неявного)* стека активации.

Для первого подхода характерно использование, во-первых, способов адресации, допускающих в памяти существование управляемых группой программ стеков магазинного типа с относительной адресацией внутри каждого стека, и, во-вторых, команд, позволяющих программе управлять своим стеком при передаче параметров, возврате адресов и выделении памяти для локальных переменных. Примером такого подхода является микропроцессор 68000 фирмы Motorola, содержащий восемь 32-битовых регистров данных и восемь 32-битовых адресных регистров. Один из адресных регистров — A7 — неявно адресуется небольшим числом команд и известен под названием *указатель стека* (SP). Обычно при организации подпрограмм еще один адресный регистр используется как *указатель кадра* (FP).

Микропроцессор 68000 располагает многочисленными способами адресации, т. е. способами формирования командами адресов памяти, а его команды отличаются высокой степенью инвариантности к способу адресации (большинство команд допускает любой способ адресации из числа возможных). Речь идет о следующих способах адресации: абсолютной, косвенной регистровой, посредством регистра базы и смещения (положительного или отрицательного), косвенной регистровой с априорным отрицательным или апостериорным положительным единичным приращением содержимого регистра, посредством индексирования и заданием адреса относительно значения счетчика команд.

Обычно в стеке активации, устанавливаемом в памяти для процесса, содержится некоторое количество кадров (последовательностей слов в стеке). Каждый кадр представляет собой запись активации для активной подпрограммы, содержащую место для локальных переменных, адрес возврата, место для записываемого на хранение

содержимого регистров и для фактических параметров, используемых для обращения к другой подпрограмме. Естественным является такое состояние, при котором указатель кадра содержит адрес начала текущего (расположенного на вершине стека) кадра или записи активации, а указатель стека содержит адрес вершины стека (последнего слова в текущем кадре). Адресация посредством регистра базы и смещения может быть использована для адресации памяти в пределах кадра (т. е. относительно указателя кадра), а косвенная регистровая адресация с априорным отрицательным или апостериорным положительным приращением содержимого регистра применяется для загрузки информации в стек или извлечения ее оттуда (т. е. запись текущего кадра в стек или извлечение его из стека).

Для анализа процесса использования подпрограмм семейст-

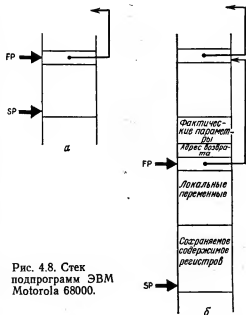


Рис. 4.8. Стек подпрограмм ЭВМ Motorola 68000.

вом микропроцессоров 68000 предположим, что в данный момент выполняется процедура А, причем текущее состояние стека соответствует тому, что изображено на рис. 4.8, а. (Способы адресации определены таким образом, что увеличение стека происходит путем расширения занимаемой им области памяти в сторону меньших по номеру адресов.) Пусть процедуре А нужно вызвать процедуру В, для чего используется оператор `CALL B(7,X)`, реализуемый четырьмя командами процедуры А. Первой является команда `MOVE` с косвенной регистровой адресацией посредством априорного отрицательного приращения содержимого указателя стека; она помещает число 7 на вершину стека. (Предполагается, что число 7 подлежит передаче по значению.) Вторая команда — команда `PUSH EFFECTIVE ADDRESS` — используется для загрузки адреса переменной X на вершину стека (эта переменная подлежит передаче путем ссылки на нее). Третья команда — команда `JUMP TO SUBROUTINE` — загружает в стек адрес возврата (адрес следующей команды) и изменяет содержимое счетчика команд. Четвертая команда, выполняемая после возврата из процедуры В, — это команда `ADD`, которая добавляет число 4 (два слова) к содержимому указателя стека и тем самым осуществляет удаление фактических параметров из стека.

Процедура В начинается с выполнения команды `LINK`. Если для локальных переменных процедуры В требуется 24 байт памяти, то эта команда имеет вид `LINK FP,24`. В процессе выполнения она, во-первых, загружает в стек текущее содержимое указателя кадра, тем самым сохраняя в данном кадре адрес предыдущего кадра; во-вторых, записывает текущее содержимое указателя стека в указатель кадра и, в-третьих, увеличивает на 24 содержимое указателя стека. По второй команде процедуры В — команде `MOVE MULTIPLE REGISTERS` — в стек загружается содержимое любых регистров, используемых этой процедурой. Теперь стек имеет вид, подобный тому, который показан на рис. 4.8, б. Локальные переменные процедуры В адресуются посредством отрицательного смещения относительно положения указателя кадра, а фактические параметры — посредством положительного смещения относительно этого же указателя. Завершается процедура В тремя командами. По первой команде — `MOVE MULTIPLE REGISTERS` — восстанавливается содержимое указанных регистров путем извлечения его из стека. По второй команде — `UNLINK` — содержимое указателя кадра помещается в указатель стека, а затем текущее содержимое вершины стека (адрес предыдущего кадра) загружается в указатель кадра. Последняя команда — `RETURN FROM SUBROUTINE` — служит для размещения содержимого вершины стека в счетчике команд.

Хотя при использовании подобного механизма функционирования подпрограмм значительная часть нагрузки по управлению подпрограммами снимается с «плеч» компиляторов, однако такому способу организации подпрограмм присущи следующие недостатки:

1. Сохранение подобного механизма как единого целого зависит от того, насколько взаимодействующие процедуры подчиняются протоколу их взаимодействий, определяемому программой. Любое отклонение от принятых соглашений (в частности, несоответствие между числом передаваемых фактических параметров и «ожидающих» их формальных, неудача в попытке использования обеими процедурами одного и того же адресного регистра в качестве указателя кадра, выделение неправильного объема памяти посредством команды LINK, неудача в попытке сохранения и восстановления содержимого необходимых регистров) может неожиданно привести к неудаче в выполнении программы.

2. При реализации данного механизма новая модель памяти не является в большей степени адекватной модели памяти, используемой языками программирования. Команды вызываемой процедуры могут модифицировать информацию в стеке, описывающую состояние системы, и имеют доступ к кадрам стеков других процессов. Таким образом, не достигается защита от несанкционированного доступа, обеспечиваемая описанными выше средствами защиты доменами.

3. Если при входе в процедуру локальным переменным необходимо задавать начальные значения, то требуются дополнительные команды.

4. Структуре стека присущ последовательный принцип организации, поэтому его максимальный размер должен быть определен заблаговременно для каждого процесса и для каждого стека должна быть заранее зарезервирована память из расчета на его максимально возможный размер.

Эти недостатки нетрудно устранить, если «спрятать» средства функционирования стека в машине и создать средства управления подпрограммами более высокого уровня в архитектуре вычислительной системы. Именно так и поступили разработчики архитектуры системы SWARD. В этом случае каждая прошедшая компилирование программная единица представлена машинным объектом, получившим название «модуль». *Модуль* — это совокупность ячеек теговой памяти и машинных кодов, представляющих одну или несколько процедур. Ячейки можно разделить на две группы. К первой группе относятся ячейки, динамически выделяемые в качестве новых элементов при вызове модуля. Вторая группа объединяет ячейки, «время

жизни» которых не зависит от обращений к модулю. (Таковыми ячейками являются, например, статические или собственные переменные.) На этапе компилирования каждой ячейке может быть присвоено начальное значение.

Для обращения к входной точке процедуры модуля используется команда **CALL**, по внешнему виду очень похожая на оператор **CALL** языков программирования. Операндами этой команды являются адрес точки входа процедуры в модуль и список фактических параметров. Команда **CALL** выполняет следующие действия:

- 1) запись на хранение состояния выполняемого модуля (например, запоминание адреса возврата в этот модуль);
- 2) выделение в памяти места для записи активации, предназначения для вызываемого модуля, и копирование туда содержимого всех ячеек вызываемого модуля, для которых хотят получить новое содержимое;

- 3) пуск вызываемого модуля на выполнение.

Первой выполняемой командой вызванного модуля обычно является команда **ACTIVATE**, которая перечисляет в качестве своих операндов имена ячеек, получающих метки (теги), как ячейки формальных параметров. Команда **ACTIVATE** заставляет машину проверять каждый формальный параметр и соответствующий ему фактический параметр на согласование по типу (для чего используются теги), а также устанавливает ссылку формального параметра на соответствующий фактический. Команда **RETURN** отменяет все назначения, выполненные командой **CALL**.

Подобный способ управления подпрограммами отличается от рассмотренного выше тем, что признаки его существования крайне незначительно отображены в архитектуре вычислительной системы. Хотя в спецификации архитектуры упоминается о наличии записи активации и стека этих записей, однако их фактический формат, местоположение и механизм взаимосвязей (например, как элементов последовательной и списковой структур) известны только тому, кто несет ответственность за практическую реализацию машины, и, следовательно, они могут изменяться при переходе от одной реализации к другой. Единственное, что доступно программисту или разработчику компилятора, — это команды **CALL**, **ACTIVATE** и **RETURN**; характер распределения памяти, формальные параметры со ссылками на соответствующие им фактические параметры и др. скрыты за «фасадом» архитектуры. Действия программы не могут разрушить целостность этого механизма. Невозможно, например, возникновение непредвиденных передач управления, модификация информации о взаимосвязи модулей и т. п. Подобный механизм предусматривает также и защиту от несанк-

ционированного доступа посредством доменов: каждый модуль — это домен.

ДОСТОИНСТВА ИСПОЛЬЗОВАНИЯ СРЕДСТВ УПРАВЛЕНИЯ ПОДПРОГРАММАМИ

Повышенная эффективность выполнения программы. Как и при других подходах к решению подобных проблем, предполагающих переключивание некоторых или всех функций управления подпрограммами на машинный интерфейс (т. е. осуществление интерфейса между подпрограммами на машинном уровне), в данном случае тоже достигается повышенное быстродействие благодаря более эффективному использованию потенциальной пропускной способности тракта «процессор — память». А поскольку рассматриваемый подход усиливает тенденцию к более глубокому структурированию (повышенной модульности) программ, то отмеченное достоинство становится еще более важным.

Уменьшение необходимого объема памяти. Как и в других ситуациях, влекущих за собой усиление роли машинного интерфейса, при использовании рассматриваемых средств управления подпрограммами сокращается размер программ и уменьшается объем генерируемых компилятором кодов, которые необходимы для реализации вызовов подпрограмм.

Расширенные возможности обнаружения ошибок. При совместном использовании теговой памяти и средств управления подпрограммами (как это имеет место в системе SWARD) машина может обнаруживать ошибки работы интерфейса. Например, она может выявить, что вместо ожидаемых пяти фактических параметров подпрограмма получает только четыре или вместо числа с плавающей точкой ей передается строка символов. Альтернативой такого решения является осуществление подобного необходимого контроля программными средствами до выполнения, например на этапе редактирования и установления связей между модулями или в процессе компилирования, как это происходит при использовании языка Ада. Хотя целесообразность такого контроля очевидна, не все ошибки интерфейса могут быть обнаружены до выполнения программы. Это объясняется следующими причинами.

Во-первых, в некоторых программах атрибуты фактических параметров не известны до наступления этапа ее выполнения. Так, процедура во время своей работы принимает решение, какие фактические параметры должны быть переданы. Во-вторых, обычно в вычислительной системе имеются интерфейсы (например, обеспечивающие запрос функций операционной системы), остающиеся не подсоединенными на этапе редактиро-

вания и установления связей. В-третьих, некоторые системы и языки программирования предоставляют возможность динамического установления связей между вызывающими и вызываемыми программными модулями на этапе выполнения. В-четвертых, в некоторых ситуациях до момента выполнения не известно, какая процедура будет вызвана конкретным оператором CALL. Это происходит, в частности, при использовании в программах на языке ПЛ/1 переменных, задаваемых оператором ENTRY.

Расширенные возможности реализации системы. Более широкое применение машинного интерфейса предоставляет разработчику процессора дополнительные возможности по организации параллельных вычислительных операций с помощью аппаратных средств. Так, можно одновременно выделить память для записи активации с целью хранения состояния вызывающей процедуры и провести проверку формальных и фактических аргументов на соответствие типов. С учетом большой вероятности весьма частого обращения вызываемой процедуры к ячейкам в своей записи активации последняя (или наибольшая возможная ее часть) может получить место непосредственно в быстродействующем буфере, или кэш-буфере. Поскольку даже в мультипроцессорной системе записи активации процесса являются локальными по отношению к процессу, процессору можно «не заботиться» о выполнении операций промежуточных записей в кэш-буфер записи активации.

ПОТЕНЦИАЛЬНАЯ АДРЕСАЦИЯ

Еще одним важным шагом в направлении сокращения семантического разрыва между машинным интерфейсом и окружающим его программным обеспечением и, следовательно, улучшения примитивной модели памяти архитектуры фон Неймана является введение принципа *потенциальной адресации* (capability-based addressing). Как и другие подобные нововведения, потенциальная адресация может быть реализована с различной степенью использования скрытых в ней возможностей. Далее рассматривается один из ее расширенных, т. е. модифицированных, видов.

Прежде всего при данном подходе отказываются от такой модели памяти, при которой последняя представляется как линейная последовательность слов. Вычислительную систему рассматривают как единый набор объектов. В определение «единый» вкладывается следующий смысл: отсутствуют какие-либо заранее предопределенные границы адресуемости, т. е. «всё» потенциально адресуемо «всеми». Термин «набор» предполагает отсутствие определенного упорядочения объектов, т. е. отсутст-

вует такое понятие, как «один объект является следующим по отношению к другому». Выше указывалось, что термин «объект» символизирует группу взаимосвязанных элементов памяти с одинаковым «временем жизни», т. е. элементы создаются и уничтожаются совместно. В зависимости от разновидности архитектуры объект может быть столь же примитивным, как и сегмент памяти (самостоятельной по своему функциональному назначению линейной по следовательности слов, внутренней структура которой по виду подобна традиционной памяти фон Неймана), или некоторой абстракцией, физическая форма которой скрыта за «фасадом» архитектуры, а назначение определяется тем преобразованием, которым она может подвергаться.

На рис. 4.9 схематически изображена рассматриваемая модель памяти. Стрелки показывают возможные обращения одного объекта к другому, т. е. свидетельствуют о наличии имен и адресов одних объектов внутри других. Следует обратить внимание, что эта модель включает все составные элементы вычислительной системы независимо от того, является ли последняя системой одного пользователя или многих. Каждый объект потенциально адресуем любым другим объектом, но такая адресация не устанавливается автоматически и не может быть создана односторонне. Это положение является основополагающим при формировании понятия «потенциальная адресация».

На данном этапе рассуждений не представляется целесообразным углубляться в природу объектов, поскольку она может меняться при переходе от одной архитектуры к другой. Однако можно воспользоваться, например, такими терминами, как объект «программа» и объект «данные». (При рассмотрении архитектуры конкретной системы появятся объекты и других типов.) Объект «программа» может представлять собой последовательность команд и, возможно, набора локальных переменных. Например, объект «программа» может соответствовать процедуре языка ПЛ/1, пакету программ языка Ада или программному модулю. Объект «данные» может отображать дина-

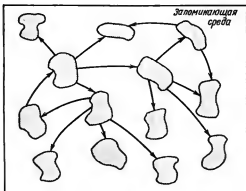


Рис. 4.9. Взаимосвязь объектов.

мически выделяемую память для новой переменной, т. е. память, «создаваемую» описателями типов переменных языка Ада. В качестве других примеров объектов можно указать на справочники адресов и файлы. Рассматриваемая модель памяти предполагает, что все объекты «программа» и объекты «данные» потенциально адресуемы друг другом, и это является основой для использования принципа разделения ресурсов программ и данных. В то же время утверждение о том, что адресуемость не является неявной и не может быть установлена односторонне, предполагает наличие средств защиты от несанкционированного доступа.

Объекты могут создаваться только с участием машины, т. е. посредством машинных команд. Имя сформированного объекта передается «создателю» объекта (имя есть результат работы машинной команды, выполняемой с целью создания объекта). Согласно принципу потенциальной адресации, имя объекта не связано с адресом возможного местоположения объекта. Имя объекта отображает адрес только в логическом, но не в физическом смысле этого слова. Информацию, необходимую для преобразования имени объекта в физический адрес его местоположения, содержит машина. Следовательно, имя объекта — это некоторая совокупность битов, назначение которых известно только машине.

При создании объектов машина присваивает им *уникальные имена* — имена, которые никогда не назначались другим объектам и не подлежат использованию для других объектов в дальнейшем. Ниже будут сформулированы доводы в пользу присвоения объектам таких имен. Конечно, употребление слова «никогда» в данном контексте нельзя назвать абсолютно точным, поскольку в этом случае имена содержали бы бесконечно большое количество битов. В действительности используются имена фиксированной (конечной) длины, однако достаточно большой для обеспечения уникальности имен в пределах ожидаемого «времени жизни» системы. Например, 48-битовые имена допускают разнообразие уникальных имен, оцениваемое величиной, которая превышает 10^{16} .

Установление потенциальной адресуемости предполагает появление одного из этих имен, и поэтому потенциальный адрес подобен обычному адресу по форме и способам использования. Функционирование механизма потенциальной адресации базируется на защите существующих потенциальных адресов и запрете создания других подобных адресов. Защита потенциальных адресов означает запрет программе модифицировать установленную потенциальную адресацию, например манипулировать потенциальным адресом одного объекта так, как будто он относится к другому объекту. Запрет создания потенциальных

адресов означает предотвращение ситуаций, позволяющих программе генерировать потенциальный адрес, подобно тому как она это делает в машинах традиционной архитектуры; программа может «завладеть» потенциальным адресом только путем создания объекта или получением его от другой программы. Эти основные правила могут быть подкреплены либо требованием хранения потенциальных адресов только в специальных «защищенных» объектах (*списках потенциальных адресов*), либо спецификацией потенциальных адресов как данных тегового типа, а затем введением ограничений на типы операций, которые могут выполняться над такими потенциальными адресами.

Перечисленные основные правила показывают, как потенциальная адресация обеспечивает защиту в модели памяти, которая предполагает потенциальную адресуемость любого объекта. Адресуемость (доступ к объектам) управляется регулированием распределения потенциальных адресов. Непременным условием возможности обращения к объекту является владение его потенциальным адресом, а подобные адреса нельзя ни создавать, ни модифицировать. Такой механизм защиты объектов от несанкционированного доступа можно сравнить со следующей физической моделью реального мира: все объекты — сумки с замками, а потенциальные адреса — ключи к ним. Тот, кто делает новый сундук, получает к нему ключ. Ключи можно дублировать («делать копии») и передавать во владение другим, но нельзя выдвигать или подделывать.

Описываемый механизм защиты может быть усовершенствован за счет сочетания потенциального адреса с директивной информацией на право доступа к объекту. В этом случае потенциальный адрес — это не просто имя или логический адрес объекта, а сочетание подобного логического адреса с индикатором права на доступ. В качестве примера введем понятия «доступ к чтению» (возможность пользоваться информацией, содержащейся в объекте), «доступ к записи» (возможность записывать информацию в объект или модифицировать хранящую там информацию), «доступ к уничтожению» (возможность уничтожить данный объект). При создании объекта машина передает тому, кто его создал, потенциальный адрес с правом полного доступа (т. е. доступа к чтению, записи и уничтожению). Предположим, что существует команда, позволяющая ограничить потенциальный адрес доступом определенного вида. Например, процесс А может создать объект Х, сделать копию полученного потенциального адреса, удалить из этой копии доступ к записи и уничтожению и передать такую копию процессу В, снабдив тем самым этот процесс возможностью потенциальной адресации к указанному объекту, однако эта возможность ограничена только доступом к чтению.

Обратимся еще раз к физической аналогии рассматриваемого механизма защиты от несанкционированного доступа: каждый сундук имеет несколько крышек. Когда открыта одна крышка, то через прозрачный экран перед нами предстает содержимое сундука (т. е. имеет место доступ к чтению); открывание второй крышки позволяет попасть внутрь сундука, но этому предшествует отключение внутреннего освещения (так реализуется доступ к записи); при открывании третьей крышки становится доступной клавиша, нажатие которой приводит к уничтожению сундука (таким путем осуществляется доступ к уничтожению). Тип доступа определяет конкретный рисунок профиля (бородки) ключа; для каждой крышки этот рисунок является своим. Рисунок может быть стерт (бородка на ключе срезана), однако добавить к нему линии новой конфигурации нельзя.

Дальнейшее улучшение рассматриваемого механизма защиты возможно путем присваивания имен отдельным элементам объекта. Если предоставлена возможность потенциальной адресации к объекту, являющемуся совокупностью отдельных («различных») элементов (например, объект — это массив или набор переменных), то естественно предположить осуществимой потенциальную адресацию к элементам. Например, если процесс А имеет потенциальный адрес к объекту X, представляющему собой массив, то этот процесс может располагать средствами создания потенциального адреса элемента $X(I)$; создав такой адрес, он может передать его процессу В. В результате процесс В получает доступ к определенному элементу объекта, но не ко всем другим элементам последнего. На практике процесс В может даже и не знать, что потенциальный адрес, которым он располагает, является элементом массива.

Подобное применение потенциальной адресации позволяет достичь высокого уровня детальности расчленения совместно используемой информации и ее защиты от несанкционированного доступа. Таким образом, потенциальный адрес можно считать состоящим из трех компонентов: индикатора права на доступ, имени объекта и указателя элемента внутри объекта. В этом случае адрес можно определить как системное имя, защищенное от несанкционированного доступа и предоставляющее право на доступ к объекту или элементу внутри объекта.

Принцип потенциальной адресации был сформулирован еще в 1966 г. [16], однако пока не оказал значительного влияния на архитектуру вычислительных машин. Он широко обсуждался при анализе операционных систем [10, 17—20] и был реализован программными средствами в экспериментальных операционных системах, таких, как CAL [21] и Hydra [22]. Разновидность потенциальной адресации нашла свое воплощение в

архитектуре вычислительной системы Plessey 250 [23], Системы 38 фирмы IBM [24], микропроцессора iAPX 432 фирмы Intel [57], экспериментальной машины CAP [25, 26], системы SWARD фирмы IBM [27, 28, 58], а также в ряде других проектов [12, 29, 30]. Опубликовано несколько интересных статей, где дан обобщенный анализ потенциальной адресации [31—35].

ПОТЕНЦИАЛЬНАЯ АДРЕСАЦИЯ В АРХИТЕКТУРЕ ЭВМ SWARD

Принцип потенциальной адресации широко используется в архитектуре машины SWARD (см. часть V), поэтому данная его реализация может служить не только иллюстрацией, но и основой для обсуждения ряда проблем, возникающих в случае применения такой адресации.

Пользуясь ранее введенной терминологией, можно сказать, что упомянутая архитектура включает в свой состав объекты

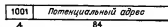


Рис. 4.10 Ячейка указателя
машины SWARD.

пяти типов. Эти объекты — абстрактные понятия, поскольку их практическая реализация архитектурой не определяется; объекты определены лишь настолько, насколько это необходимо для описания операций, применимых к ним. Объекты четырех типов — модули, порты, память данных и процесс-машины — создаются и уничтожаются программами явным образом; объекты пятого типа — записи активации — создаются и уничтожаются средствами управления подпрограмм неявным путем.

Одна из разновидностей теговых ячеек этой архитектуры получила название *указатель* (рис. 4.10). Содержимое ячейки «указатель» — это потенциальный адрес или данные «неопределенного» значения. Назначение отдельных битов потенциального адреса архитектурой не определяется; однако потенциальный адрес включает в себя четыре индикатора доступа и логический адрес объекта или элемента внутри объекта. Индикаторы доступа имеют следующие названия: чтение, запись, уничтожение и копирование. Первые три индикатора являются потенциальными ограничителями типов операций, которые могут выполняться над элементами, адресуемыми с помощью потенциальных адресов; четвертый индикатор сам «обслуживает» потенциальный адрес. Если воспользоваться для сравнения какой-либо аналогией реального мира, то можно рассмотреть, например, ситуацию, когда некое лицо А желает дать ключ лицу В и в то же время хочет предотвратить возможное изготовление последним дубликата этого ключа (который мог бы попасть в руки

третьего лица). Возможным решением является гравировка на ключе для лица текста «не дублировать», т. е. указания, которое соблюдают все изготовители дубликатов ключей. Следовательно, отсутствие разрешения на дублирование запрещает владельцу потенциального адреса копировать потенциальный адрес в другой ячейке указателя.

Для рассматриваемой архитектуры действительно следующее правило: при создании объекта формируется потенциальный адрес, содержащий право на доступ всех видов: чтение, запись, уничтожение, копирование. В то же время предоставляет команда, позволяющая лишить потенциальный адрес прав на некоторые (но не все) виды доступа.

Чтобы не нарушать общности принципов программирования, ячейками «указатель» можно пользоваться так же, как адресами в традиционных вычислительных системах (принимая во внимание, что над потенциальными адресами выполнение арифметических операций недопустимо). Ячейки «указатель» рассматриваются как ячейки определенного типа, отличного от других типов. В программе с указателями можно обращаться как с переменными: они могут существовать в пределах действия структуры данных пользователя и передаваться как фактические параметры. Возвращаясь к рассмотренной выше аналогии, можно сказать, что имеется возможность хранить ключи или связку ключей и другие «ценности» в запертых сундуках. Как уже было отмечено, механизм защиты потенциальных адресов в нетеговой запоминающей среде обычно предполагает возможность их размещения только в специально отведенных для этой цели объектах (списках потенциальных адресов). Именно это имеет место в вычислительной системе САР Кембриджского университета, разработчики которой пришли к выводу о том, что использование отдельных списков потенциальных адресов (не размещаемых в упомянутых выше объектах) является недостатком системы [26].

Выше упоминалось, что при описании архитектуры системы назначение отдельных битов потенциальных адресов (рис. 4.10) архитектурой системы не указывается (программа все равно их никогда непосредственно не «видит», а отсутствие предопределенного архитектурой назначения этих битов предоставляет свободу действий при практической реализации системы). Для знакомства с воплощением на практике подобной архитектуры полезно рассмотреть конкретный вариант использования таких 84 бит. Например, 4 бит содержат директивную информацию, 30 бит используются для описания уникального имени объекта (называемого SON—system object name—имя объекта системы). Если потенциальный адрес предназначен для обращения к элементу внутри объекта, то 24 бит используются как

смещение тега элемента относительно начала объекта и еще 24 бит — в качестве смещения содержимого элемента или его значения. (Необходимость в двух смещениях объясняется тем, что данные элемента и его тег не обязательно размещаются в смежных областях памяти, являясь, например, элементом массива.) Оставшиеся неописанными два последних бита играют роль специальных индикаторов. Если максимальный размер физически реализуемого адресного пространства равен 2^{54} , то в нем может размещаться не более 2^{30} объектов при условии, что размер каждого из них не превышает 2^{24} .

Каждому объекту необходимо присвоить свое, присущее только ему имя, чтобы предотвратить программу от создания объекта, записи на хранение его потенциального адреса, уничтожения объекта и возможного злоумышленного использования его потенциального адреса в дальнейшем для обращения к объекту, которому присвоено имя ранее существовавшего объекта. Очевидно, что отведенные 30 бит не являются неисчерпаемым источником уникальных имен, однако на практике этого оказывается достаточно по следующим двум причинам. Во-первых, при скорости создания объектов, равной 10 объект/с (записям активации, хотя и считающимся объектами, имя обычно не присваивается), упомянутый источник не может быть израсходован машиной по крайней мере в течение 3,5 лет. Во-вторых, даже если прибегать к присваиванию уже использованных имен, вероятность нарушения защиты доступа мала, поскольку речь идет об архитектуре машины с теговой памятью (см. часть V).

Получив возможность использования потенциального адреса, машина должна обладать способностью переводить имя объекта в его местоположение в физически реализуемой памяти. Это осуществляется посредством карты взаимосвязи имен объектов системы и физического местоположения этих объектов. Потенциальные адреса — это только некоторая форма адресов взаимосвязи объектов; использование этих адресов для обращения к отдельным элементам в пределах данного объекта не является обязательным. Ранее, при описании средств управления подпрограммами и организации санкционированного доступа с использованием доменов, указывалось, что программа для машины SWARD разделяется на модули, каждый из которых располагает своим собственным адресным пространством, одна часть которого расположена в самом модуле, а другая — в текущей записи активации. Для доступа к своим собственным операндам в этом адресном пространстве команды не пользуются потенциальными адресами; последние нужны только для обращения к внешнему адресному пространству. Например, если необходим доступ к массиву, который определен как не-

зависимый объект памяти данных, команда будет обращаться к ячейке «косвенный доступ к данным» в своем адресном пространстве, которое содержит соответствующую ячейку «указатель» с потенциальным адресом объекта «массив». Этот механизм подобен механизму функционирования базированных переменных в языке ПЛ/1 и переменных доступа в языке Ада.

В дополнение к командам, создающим объекты, в рассматриваемой архитектуре имеется команда `COMPUTE-CAPABILITY`, которая, получив адресуемый операнд, формирует его потенциальный адрес. Если, например, имеется потенциальный адрес объекта «массив», то с помощью этой команды можно создать потенциальный адрес для конкретного элемента массива. Аналогично можно сформировать потенциальный адрес для локальной переменной. Еще раз отметим, что хотя записи активации — это объекты, при их создании не происходит автоматического присвоения им имен; это делается с целью предотвращения исчерпывания источника уникальных имен для объектов, поскольку записи активации создаются намного чаще, чем объекты других типов. Запись активации нуждается в уникальном имени только в том случае, когда программа выполняет команду `CREATE-CAPABILITY` для локальной переменной. Следовательно, машина присваивает имя записи активации только при наличии и в момент выполнения указанной команды, присваивающей потенциальный адрес локальной переменной в записи активации.

Как и следовало ожидать, введение потенциальной адресации вызывает новые проблемы, большинство которых имеет прямые аналоги с моделью реального мира в виде запираемых сундуков и ключей к ним. Перейдем к рассмотрению этих проблем и их решению в машине `SWARD`. Одна проблема связана с предотвращением нежелательного копирования потенциального адреса, а также его «злоумышленной» передачи процессом или подпрограммой «третьей стороне». Эта проблема уже ранее упоминалась и решается включением директивной информации о копировании в потенциальный адрес.

Другая проблема может быть определена как *глобальное удаление потенциального адреса* или как проблема «нового замка». После распределения потенциальных адресов объекта нет возможностей для пересмотра решения (т. е. отказа от введенных потенциальных адресов), за исключением уничтожения объекта и его повторного создания с новым уникальным именем. В аналогичной ситуации для модели реального мира речь идет о желании сделать непригодными все имеющиеся в употреблении ключи, поскольку, во-первых, владение сундуком передано в другие руки, во-вторых, существуют подозрения, что один или несколько владельцев ключей пользуются содержи-

мым сундука недозволенным образом, и, в-третьих, имеет место неопределенность, связанная с отсутствием информации о том, как были распределены ключи. Решение проблемы физической модели состоит в смене замка.

В машине SWARD для той же цели используется команда **CHANGE-LOGICAL-ADDRESS**. Получив потенциальный адрес объекта (с директивной информацией «Разрешено уничтожение объекта») в качестве операнда, эта команда заставляет машину «забыть» текущее имя объекта, присвоить ему новое имя и вернуть адресату потенциальный адрес с новым именем. В дальнейшем все попытки обращения посредством потенциального адреса со старым именем объекта завершаются сообщением об ошибке в программе.

Еще одна проблема, порождаемая механизмом потенциальной адресации, может быть сформулирована как проблема *селективного удаления потенциальных адресов*. Например, вначале ключи были выданы 10 лицам, а позже принимается решение лишить лицо D такого ключа. В данном случае физическая модель оказывается неудобной для объяснения аналогиями, возможное решение не носит характера прямого действия: вместо того чтобы вручить ключ от самого сундука, дают ключ от другого сундука, содержащего ключ к первому сундуку. Лицо или группу лиц можно лишить прав пользования ключом путем уничтожения одного из сундуков, хранящих ключи, или посредством смены замка в таком сундуке.

В машине SWARD эта проблема решена введением косвенных потенциальных адресов, которые являются не дополнительным типом данных, а содержимым любой ячейки «указатель», порождением командой **COMPUTE-INDIRECT-CAPABILITY**. Эта команда располагает двумя операндами, каждый из которых должен быть ячейкой «указатель». Логический адрес второго операнда вычисляется и записывается в первый операнд, помечая его как косвенный потенциальный адрес.

Программа, использующая средства обычной потенциальной адресации, не может отличить косвенный потенциальный адрес от обычного адреса. Физически косвенный адрес является ссылкой на другую ячейку «указатель», но логически (по сути адресации) это ссылка на тот же указатель. Любое обращение посредством косвенного потенциального адреса имеет то же действие, которое имеет место при использовании прямой потенциальной адресации. Операции, которые могут быть выполнены над содержимым ячеек типа «указатель» (например, копирование их содержимого в другие подобные ячейки, отмена разрешения на доступ определенного вида), можно выполнять и над указателями, содержащими косвенные потенциальные адреса.

Косвенная потенциальная адресация находит несколько применений, одним из которых является защита от несанкционированного доступа посредством селективного удаления потенциальных адресов. Пусть процесс А намеревается предоставить процессу В доступ к объекту Х, но при этом хочет сохранить за собой возможность в любое время отменить право на этот доступ. Предоставляя В косвенный потенциальный адрес указателя (в адресном пространстве А), содержащего ссылку на Х, А может модифицировать в любое время этот указатель, лишив тем самым В доступа к Х.

Другой очевидной проблемой рассматриваемой модели реального мира является ситуация «потери ключей». В случае потенциальной адресации это означает потерю всех потенциальных адресов данного объекта. Например, по завершении создания объекта ошибочно выполняется запись в ячейку единственного указателя объекта. Родственной проблемой можно считать и потерю необходимых прав на доступ к объекту. Предположим, что ни один из потенциальных адресов объекта не имеет разрешения на уничтожение объекта, а это означает, что последний навсегда оккупировал адресное пространство системы.

Существуют по крайней мере две точки зрения на пути решения подобной проблемы. В соответствии с одной из них, формулируемой для системы Intel 432 (но не для системы SWORD), объект, не имеющий к себе ссылок, рассматривается как подлежащий уничтожению: при этом предполагается существование в машине средств сбора «программного мусора», опознающих и уничтожающих такие объекты. Разработчики архитектуры SWORD не нашли решения, которое не было бы крайне неэффективным или не нарушало функционирования средств защиты архитектуры. Однако соблюдение требований, связанных с защитой системы, — не единственное препятствие на пути удовлетворительного решения рассматриваемой проблемы. Единственным связующим звеном между «осведомленностью» машины об объектах и восприятием этих же объектов программами являются имена объектов в потенциальных адресах. При исчезновении последних программы и машина теряют единственное средство связи.

Используя некоторые возможности архитектуры машины, можно по крайней мере частично устранить затруднения, лежащие на пути решения описанной проблемы. Во-первых, команды, создающие объекты, позволяют указывать, подлежит ли объект автоматическому уничтожению машиной по завершении процесса создания объектов. Это используется при решении проблем, связанных с излишней продолжительностью «времени жизни» в системе большого числа объектов, предположительно временных, но вовремя не уничтоженных вследствие аномаль-

ного завершения процесса. Во-вторых, программам, создающим постоянные (долговременного пользования) объекты, рекомендуется (но не вменяется в обязанность) хранить свои потенциальные адреса в справочниках операционной системы. В-третьих, практически реализованный прототип машины данной архитектуры содержит средства сбора «программного мусора», которые позволяют обслуживающему персоналу осуществлять поиск, возврат или уничтожение потерянных объектов.

Еще одна проблема, связанная с использованием потенциальной адресации, может быть названа проблемой «определения нужного ключа из числа возможных». В аналогичной модели реального мира это эквивалентно отсутствию информации о признаках ключа в связке ключей или такого ключа, который найден на земле. Речь идет о ситуациях, в которых машина располагает полезной информацией, не предоставленной программам. Применительно к машине SWARD указание можно сформулировать следующим образом.

1. В программе отсутствуют явно описываемые средства прочтения директивной информации, содержащейся в потенциальном адресе (например, нет средств проверки, содержит ли конкретный потенциальный адрес разрешение на запись).

2. По имеющемуся потенциальному адресу нельзя непосредственно определить тип соответствующего объекта или элемента этого объекта: является ли этот объект программным модулем, точкой входа в модуль, ячейкой памяти данных, портом или чем-либо другим.

3. Машина располагает определенной информацией о состоянии объектов, которая недоступна программам. Примерами такой информации, необходимой программе, являются ответы на следующие вопросы: подлежит ли данный объект автоматическому уничтожению по завершении процесса? Активен ли данный модуль? Имеются ли ждущие обслуживания элементы в очереди данного порта?

Данная проблема решена включением в набор команд машины команды DESCRIBE-CAPABILITY. Располагая указателем и массивом в качестве операндов, команда возвращает в заданный массив необходимую информацию об объекте, описываемом потенциальным адресом, который содержится в указателе. Такой информацией может являться, например, класс адресата потенциального адреса и тип доступа, которым располагает адресат. Если потенциальный адрес обеспечивает обращение ко всему объекту, то команда передает информацию о его состоянии. Только часть подобной информации не зависит от типа объекта. Описываемая команда не передает информацию о содержимом объекта. Так, в случае объекта «данные» она не описывает ни тип ячеек, ни их содержимое.

ДОСТОИНСТВА ПОТЕНЦИАЛЬНОЙ АДРЕСАЦИИ

Надежность защиты от несанкционированного доступа. Потенциальная адресация является простым средством защиты информации от несанкционированного доступа. При ее использовании исчезает необходимость в таких средствах защиты, как выделение состояний системы (например, состояния «супервизор») или введение ключей защиты областей памяти. Директивная информация в форме спецификации в потенциальном адресе обеспечивает несколько уровней доступа к объектам, т. е. возможности, часто отсутствующие в традиционных вычислительных системах. Будучи принципиально простым механизмом, потенциальная адресация обладает двумя важными свойствами: во-первых, предоставляет доступ только к тому адресу, к которому имеет место обращение, и, во-вторых, не допускает подделки и несанкционированного генерирования потенциальных адресов, а также модификации их значений.

Еще одно достоинство потенциальной адресации как средства защиты выявляется в ситуациях, когда специально принимаются меры к временной отмене действия этого средства, например всевозможными программными «трюками» пытаются заставить операционную систему предоставить программе пользователя состояние «супервизор». В подобных случаях традиционная система становится полностью незащищенной от несанкционированного доступа. В системе с потенциальной адресацией это приводит к появлению неразрешенного к использованию (в нормальной ситуации) потенциального адреса, в результате чего незащищенным окажется только тот объект, на который этот адрес ссылается, а не вся система.

Более простое совместное использование данных и программ. Пользуясь в простейшем случае моделью памяти как единственным набором объектов, потенциальная адресация позволяет отказаться от традиционного механизма совместного использования данных и программ [31]. Все, что нужно сделать для этого в данном случае, сводится к предоставлению другой программе потенциального адреса объекта, которым до сих пор пользовалась только данная программа.

Расширенные возможности уточнения границ областей санкционированного доступа и совместного использования данных и программ. Потенциальная адресация позволяет устанавливать границы доменов защиты и объектов совместного использования посредством терминов, информативных для программы, что делает эти границы более гибкими, чем в машинах традиционной архитектуры. Например, программа А может вычислить потенциальный адрес для переменной X и передать его про-

грамме В, предоставляя только этой программе доступ к переменной Х.

Унификация средств архитектуры машины. Объединяя часто разьединенные средства адресации и защиты от несанкционированного доступа и устраняя необходимость в таких «половинчатых» решениях введением состояний «привилегированная команда», потенциальная адресация способствует достижению значительной унификации принципов и средств архитектуры машины. Представление информации в виде объектов позволяет проектировать вычислительную систему, придерживаясь принципа унификации межобъектных связей (т. е. можно отказаться от практики применения одного принципа при организации взаимосвязи программных модулей, другого принципа для установления связи программы с данными и т. д.).

Более совершенные средства обнаружения ошибок. Благодаря использованию в потенциальных адресах уникальных имен объектов машина способна обнаруживать программную ошибку, известную под названием *повисшая ссылка*. Такая ошибка возникает, когда «время жизни» адреса превосходит «время жизни» объекта, на который этот адрес ссылается. Предположим, что выполняется следующий оператор языка ПЛ/1: $P = ADDR(X)$. Возможны ситуации, когда переменная Х уже исчезла, а переменная Р еще сохраняется. Например, Х — локальная переменная в процедуре, а Р — возвращенный параметр, причем по завершении процедуры исчезновение Х логично, или же Х — динамически определяемая переменная, действие которой в дальнейшем отменяется. В современных вычислительных системах Р — это просто адрес, при использовании которого для обращения к памяти после уничтожения Х возникает непредсказуемый результат. В системе с потенциальной адресацией значение Р — это потенциальный адрес, при попытке использования которого после уничтожения Х регистрируется невозможность вычисления фактического адреса, а следовательно, происходит обнаружение ошибки.

ОДНОУРОВНЕВАЯ ПАМЯТЬ

При использовании современных вычислительных систем программист сталкивается с отсутствием единообразия и непрерывности в модели памяти. Машина не может «скрыть» от него различия, существующие в адресации памяти, ее функционировании и сохранении данных. Так, данные в основной памяти адресуются одним способом (например, значениями адресов, образующими одномерную монотонно меняющуюся числовую последовательность), а данные на поверхности магнитного диска — совершенно другим способом (совокупностью значений

номеров дискового, дорожки, записи и значением линейного смещения от начала записи). Базовые операции, такие, как сложение, сравнение, пересылка или копирование, не определены вместе для всей модели памяти. Поэтому программист должен явным образом указывать пересылку данных в запоминающую среду того типа, для которого необходимая функция определена. Кроме того, программист должен помнить, что в основной памяти информация удерживается только до завершения процесса, в то время как во внешней памяти она содержится до тех пор, пока не будет уничтожена явным образом.

Указанное выше требует от программиста (а им часто является прикладной программист) понимания, что памяти машины присуща отмеченная нерегулярность организации. На программиста возлагается обязанность явным образом задавать необходимые перемещения данных между различными запоминающими средами (т. е. выполнять операции ввода и вывода) с целью получения рабочей среды с требуемыми характеристиками адресации памяти, ее функционирования и сохранения данных. Кроме того, поскольку объем основной памяти часто ограничен, программисту приходится прибегать к запоминающим средам с другими принципами организации данных (например, файловая структура), которые представляют определенные возможности для передачи данных между программами. Отсюда можно сделать вывод, что наличие запоминающих сред с различной организацией данных вносит свой вклад в высокую стоимость программирования.

Перечисленные выше свойства памяти традиционной машины являются причиной увеличения разнообразия типов интерфейса между программами, что препятствует достижению единообразия и модульности структуры программного обеспечения. В настоящее время программа может получать входную информацию двумя путями: через список фактических параметров (т. е. в виде структурированных или неструктурированных данных, расположенных в основной памяти) или посредством операций ввода в виде данных внешней памяти (имеющей, например, файловую структуру). Поскольку различия между этими двумя путями ввода информации слишком велики и очевидны, они редко одновременно сочетаются в одной программе. Если программный модуль (программа) А написан с указанием ввода данных через список фактических параметров, то такой модуль может оказаться неудобным для использования в тех случаях, когда обработке подлежит большое количество входных данных (объем которых может превышать емкость основной памяти) или когда желательно выполнение программного модуля А независимо от других программных модулей. Аналогично, если программный модуль В написан с указанием операций

ввода данных как файла, этот модуль может оказаться неудобным для использования в таких ситуациях, когда подлежащие обработке данные размещены в другом программном модуле или когда файловая структура этих данных отличается от приемлемой для модуля В структуры данных способом представления в запоминающей среде (как, например, файлы на магнитной ленте отличаются от файлов на магнитных дисках).

Решением указанных проблем является использование в архитектуре машины простого принципа одноуровневой памяти. (Определение «простой», справедливое применительно к разработке архитектуры машины, не обязательно применимо к этому принципу при его воплощении в реальной машине.) Указанный принцип предполагает унификацию всех разновидностей запоминающей среды вычислительной системы таким образом, чтобы при формировании интерфейса машины с программными средствами все разновидности физически реализуемых запоминающих устройств были представлены одинаковыми средствами адресации, наборами выполняемых функций и характеристиками представления данных в памяти. Например, условно изображенная на рис. 4.9 модель запоминающей среды может представлять в таком случае не только основную память, но и все одноуровневое запоминающее пространство. Функциональные элементы традиционной памяти — файлы — становятся теперь объектами (элементами одноуровневой памяти) и адресуются согласно общим правилам обращения к объектам. Если же теперь по соображениям экономии средств или достижения требуемого быстродействия вычислительная система имеет иерархическую структуру памяти в виде совокупности запоминающих устройств различных стоимости и быстродействия, то функция перемещения данных между запоминающими устройствами различных уровней иерархии возлагается на машину, а не на программиста.

Одноуровневая память во многом схожа с виртуальной памятью, однако имеются и различия. Во-первых, виртуальная память (в общепринятом понимании этого термина) не предполагает унификацию отдельных, составляющих ее запоминающих сред; как правило, она предназначена просто для расширения тех функциональных возможностей системы, которые обычно ограничены основной памятью. С этой точки зрения принцип одноуровневой памяти можно рассматривать как распространение принципа виртуальной памяти на все запоминающее пространство системы. Малопривлекательную для этих целей линейную модель такого пространства желательно заменить описанной выше моделью в виде набора объектов с потенциальными адресами (рис. 4.9). Во-вторых, при работе с виртуальной памятью обычно с завершением выполнения программы «исчезает» и об-

служивающий ее набор запоминающих сред; то же самое происходит при выключении питания системы. Совсем другая картина возникает при использовании одноуровневой памяти. Например, если желательно сохранить до следующего дня данные программы, расположенные в объекте «массив», то нет необходимости создавать временный файл для их хранения; массив просто удерживается в памяти. Отметим, что этот же принцип применим и к объекту «программа». Все зависит только от того, как для такой ситуации разработчик архитектуры системы определил понятие «загрузка программы». Например, статические или собственные переменные могут сохранять свои значения между отдельными шагами выполнения программы. Это предоставляет не только простой механизм запоминания данных на время перехода от одного шага выполнения к другому, но и вызывает необходимость пересмотра определения некоторых принципов, лежащих в основе языков программирования.

ДОСТОИНСТВА ОДНОУРОВНЕВОЙ ПАМЯТИ

Сравнительно низкая стоимость программного обеспечения. Значительная часть средств, расходуемых на разработку программы, затрачивается на решение сложных проблем ввода-вывода и (или) управление взаимодействием запоминающих сред разных уровней иерархии памяти. Последняя функция выполняется программистом с помощью программных средств. Одноуровневая память не избавляет программиста от всех проблем ввода-вывода в программе, например организации ввода-вывода для дисплеев и аналого-цифровых преобразователей, вывода для печатающего устройства. Однако, если говорить о языке программирования, то использование одноуровневой памяти позволяет изъять из программ и языков много сложных конструкций, связанных с организацией ввода-вывода. Например, вместо явной записи в программе таких операций над файлами, как READ, WRITE, GET и PUT, достаточно определение файла как массива или вектора, что предполагает в нужный момент выполнение указанных операций. Кроме того, одноуровневая память способствует достижению более совершенной модульности программы благодаря единообразию формы всех запоминающих сред, между которыми программа осуществляет пересылку данных.

Может показаться, что все упомянутые достоинства одноуровневой памяти достигаются при использовании в современных вычислительных системах разработанных в последнее время пакетов управления базами данных. Однако это справедливо лишь отчасти. Во-первых, об этом можно говорить примени-

тельно лишь к большим системам, обладающим таким пакетом, и только в случае использования последних. В меньших по размеру системах (на базе мини-ЭВМ или микропроцессоров) такие пакеты, как правило, отсутствуют. Во-вторых, наличие пакетов не избавляет программиста полностью от проблем организации ввода-вывода: в программе осуществляется описание интерфейса записи данных в базу и копирования их оттуда. В-третьих, в настоящее время базы данных не охватывают файлы всех типов; они ориентированы на детально структурированные совокупности данных с относительно неизменной (статичной) формой структуры и оказываются весьма неэффективными при работе с небольшими динамическими наборами данных или с неструктурированными данными.

Независимость принципов организации памяти от реализующей ее технологической базы. Очевидной характеристикой одноуровневой памяти является то, что конкретные запоминающие устройства, реализующие эту память, и иерархия их взаимоотношений скрыты за интерфейсом машины. Благодаря этому программы легче «переносить» с одной системы на другую; они становятся менее зависимыми от конкретной конфигурации вычислительной системы. При этом разработчик машины свободен в выборе (или изменении) иерархии взаимосвязей запоминающих устройств, скрытых от программ интерфейсом машины. Поскольку в настоящее время технология производства запоминающих устройств быстро меняется, последнее из перечисленных выше обстоятельств представляется наиболее важным. Хотя общая тенденция технологии производства запоминающих устройств различных типов такова, что стоимость их падает, а быстродействие растет, соотношение между этими показателями не сохраняется постоянным; кроме того, все время появляются запоминающие устройства с новыми характеристиками.

В настоящее время проектировщику вычислительной системы приходится принимать во внимание ряд факторов. Во-первых, следует учитывать уменьшение разницы в стоимости магнитной запоминающей среды (например, памяти на магнитных дисках) и полупроводниковой. Во-вторых, имеет значение сходство по многим параметрам памяти на магнитных доменах (в виде сдвигающих регистров) и вращающихся магнитных дисках при наличии у памяти первого типа таких дополнительных свойств, как способность прекращать и возобновлять пересылку данных одновременно с изменением состояния механизма чтения-записи. Кроме того, существенно наличие высокоскоростных маломощных и дешевых полупроводниковых запоминающих устройств произвольного доступа и энергонезависимых полупроводниковых запоминающих устройств произвольного доступа.

Унификация средств архитектуры машины. Принцип одноуровневой памяти не только позволяет программам «рассматривать» различные запоминающие среды как память одного типа — единой формы представления данных, но и распространяет такой подход к описанию возможностей всей системы (а не только в контексте с описанием функционирования основной памяти). Например, если в архитектуре системы используется принцип, именуемый «теговая память», он распространяется на всю иерархию запоминающих сред. Атрибут «теговый» применим теперь и к совокупностям данных, которые ранее определялись как файлы.

Глобальная оптимизация. В традиционных системах каждая программа, выполняющая операции ввода-вывода, управляет иерархией запоминающих устройств системы в соответствии со своими собственными критериями. Более того, как правило, процессор осуществляет независимое управление кэш-памятью, а операционная система — процессом «перелистывания» страниц памяти. Весьма вероятно, что с позиций эффективности функционирования системы в целом эти локальные и независимые друг от друга операции далеки от оптимальных. Такие действия, как двойное буферирование, возможно совершенные для локальной оптимизации, могут оказаться весьма далекими от оптимальных в глобальном смысле этого определения. Вот почему «сокрытие» иерархии запоминающих сред от программ и управление ею единым механизмом может способствовать оптимизации этого процесса и работ системы в целом. Как и прежде, имея этот механизм за «фасадом» машинного интерфейса, разработчик вычислительной системы располагает большей свободой в выборе средств ее реализации, например используя средства параллельного выполнения вычислений.

ПРОБЛЕМЫ, ВОЗНИКАЮЩИЕ ПРИ ИСПОЛЬЗОВАНИИ ПРИНЦИПА ОДНОУРОВНЕВОЙ ПАМЯТИ

Реализуя практически принцип одноуровневой памяти, разработчик вычислительной системы сталкивается с некоторыми новыми проблемами. Время покажет, разрешимы они или их следует отнести к недостаткам, присущим одноуровневой памяти.

Реализация иерархии запоминающих устройств. Принцип одноуровневой памяти предполагает наличие иерархии запоминающих устройств, содержащих все данные и программы системы. Эта иерархия, возможно, состоит из многих уровней запоминающих сред с различными значениями таких параметров, как стоимость и быстродействие. К настоящему времени опыт построения упомянутых иерархий еще мал (кэш-память, память произвольного доступа, память на магнитных дисках со стра-

ничной организацией) [36]. Необходимо проведение значительных исследований с целью определения числа уровней иерархии, стратегии перемещения данных, размера одновременно пересылаемой информации (объект целиком, часть объекта, страница фиксированного размера), стратегии «сквозной записи» (т. е. выяснение, необходимо ли отражение выполнения операций чтения в постоянной области объекта), требований к энергонезависимости памяти и рекомендаций по организации мультипроцессорной работы или работы с распределенными процессорами.

Сохранение чистоты принципа одноуровневой памяти. Хотя, согласно изложенному выше, «сокрытие» от программ всех характеристик иерархии запоминающих сред (и даже самого факта существования такой иерархии) желательно, остаются спорными вопросы о практичности такого решения во всех ситуациях и необходимости предоставления определенным программам средств осуществления локальной оптимизации. Речь идет о средствах, которые могли бы предложить, рекомендовать или директивно приказать программе следующее: перемещать сквозь иерархию запоминающих сред набор объектов, связанных «логикой» решаемой задачи; удерживать определенный объект в течение указанного периода времени в запоминающей среде наивысшего уровня иерархии; перемещать сквозь иерархию объекты (например, в таких ситуациях, как следующая: «я готов начать использование этого объекта»). Остается нерешенным вопрос об определении того, какие именно средства достаточны для поставленной цели и как представить их в архитектуре, соблюдая при этом требование максимально возможной независимости принципов действия этих средств от путей их реализации.

Восстановление памяти. Предполагая неизбежными нарушения в работе запоминающих устройств (например, повреждение головок записи и считывания, разрушение оксидного покрытия, искажение записи излучением альфа-частиц), естественно предположить, что применение принципа одноуровневой памяти, скрывающего от программ пользователя наличие иерархии запоминающих сред и не позволяющего анализировать их структуру, должно затруднить процесс восстановления памяти. Это предположение основано на том, что процесс восстановления памяти существующих вычислительных систем требует привлечения умственных способностей человека, т. е. анализа им картины размещения информации в реальных, физических реализованных областях памяти (например, может появиться необходимость выяснить, какие файлы находятся на тех или иных дорожках диска) и анализа содержания этой информации (так, может возникнуть потребность найти избыточные данные или

определить способ реконструкции утерянных данных). Возможным решением рассматриваемой проблемы является применение самоопределяемых единиц пересылаемой информации (таких, как сегмент, страница), позволяющих их восстановление даже при разрушении системного справочника или нарушении в работе иерархии запоминающих сред, и создание служебного интерфейса для общения человека-оператора с памятью системы.

Транспортабельность объектов. Вычислительная система с одноуровневой памятью является замкнутой, т. е. ее объекты нельзя перенести в другие системы (объекты не обладают свойством транспортабельности). Поэтому задача переноса объекта из системы А в одноуровневую память системы В нуждается в специальном рассмотрении. Пользуясь средствами современной вычислительной техники, объект можно поместить в специальный пакет дисков. Однако при использовании одноуровневой памяти местоположение объекта неизвестно; он может быть размещен даже по частям в различных запоминающих устройствах.

В случае применения потенциальной адресации также возникают проблемы транспортабельности объектов. Потенциальный адрес объекта в одной системе теряет всякий смысл при переносе объекта в другую систему.

Организация ввода-вывода устройств без памяти. Хотя применение одноуровневой памяти и позволяет отказаться от традиционных принципов организации ввода-вывода файлов, модель такой памяти непригодна для описания ввода-вывода в устройствах без памяти и устройствах или запоминающих средах, для которых не может использоваться произвольный или прямой доступ к памяти. Примерами устройств ввода-вывода без памяти являются экраны электронно-лучевых трубок, клавишные панели, устройства чтения перфокарт, устройства печати, линии связи. Вообще говоря, и для них можно найти адекватное представление в модели памяти. В частности, в некоторых микропроцессорных системах получил распространение принцип организации ввода-вывода по аналогии с обращением к памяти. Однако такое представление не является непосредственным и может служить источником дополнительных неопределенностей при функционировании системы. Например, такая запоминающая среда, как магнитная лента, непригодна для использования в качестве составной части одноуровневой памяти, поскольку для нее необходимы методы последовательного доступа. Следовательно, одноуровневая память не всегда позволяет отказаться от использования в архитектуре машины принципа ввода-вывода.

ОДНОУРОВНЕВАЯ ПАМЯТЬ В СИСТЕМЕ 38 ФИРМЫ IBM

Поскольку Система 38 фирмы IBM [60], по всей видимости, является первой промышленно выпускаемой вычислительной системой, в которой реализован принцип одноуровневой памяти, применение на практике этого принципа рассмотрим на примере данной системы. Однако прежде всего следует ознакомиться со структурой этой неортодоксальной вычислительной системы.

Современные варианты построения Системы 38 имеют двух-

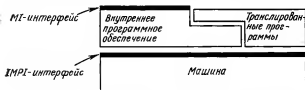


Рис. 4.11. Двухуровневая архитектура Системы 38.

уровневую архитектуру (рис. 4.11). Архитектура внешнего уровня — так называемая *MI-архитектура* — является доступной для программиста частью архитектуры этой вычислительной системы. На указанном уровне осуществляется компилирование программ. Однако этот уровень нельзя назвать интерпретирующим. Когда программы загружены, средства внутреннего программного обеспечения транслируют их на внутренний, более низкий, интерпретирующий уровень, именуемый *IMPI-архитектурой*, ниже которого располагается машина с микропрограммным управлением. Архитектура этого уровня программисту непосредственно недоступна, благодаря чему изготовители Системы 38 имеют возможность модифицировать IMPI-архитектуру при создании последующих вариантов построения системы. Хотя MI-интерфейс (рис. 4.11) и не является интерпретирующим, внутреннее программное обеспечение создает у программиста такую иллюзию (т. е. программные ошибки и информация о состоянии системы сообщаются в терминах MI-интерфейса).

Отметим, что необычная структура данной вычислительной системы затрудняет определение ее архитектуры. Строго говоря, IMPI-интерфейс отображает архитектуру аппаратных средств машины, а MI-интерфейс — программное обеспечение. Однако, поскольку для пользователя доступен только MI-интерфейс и на данном уровне передают результаты своей работы компиляторы, а также в связи с тем, что система создает иллю-

люзию выполнения программ именно на этом уровне, можно считать, что МІ-интерфейс определяет архитектуру системы.

Модель памяти МІ-интерфейса подобна той, которая представлена на рис. 4.9. Это — ориентированная на объекты модель, использующая потенциальную адресацию и представляющая собой одноуровневую память.

Одноуровневый принцип хранения информации в МІ-интерфейсе базируется на следующих положениях.

1. Имеются «суперобъекты», называемые объектами типа «группа доступа»; каждый объект этого типа может включать в себя один или несколько объектов других типов. Эти объекты используются для управления перемещением данных между адресатами иерархии запоминающих устройств. В объект «группа доступа» можно поместить объекты, к которым будет осуществляться одновременное обращение (например, объекты, совокупность которых образует определенную программу). В результате этого достигается следующий эффект: при обращении к одному объекту группы доступа другие ее объекты испытывают тенденцию быть перемещенными в то же самое время в более быстродействующее запоминающее устройство.

2. Явно задаваемая команда SET-ACCESS-STATE, которой располагает система, позволяет программе запрашивать перемещение объекта в основную память и удаление его из этой памяти. Такое перемещение объекта может быть определено как синхронное (выполнение команд вычислительного процесса не продолжается до тех пор, пока не будет удовлетворен запрос) или асинхронное.

3. Наличие команды ENSURE-OBJECT дает возможность программе заставлять систему сохранять текущую копию объекта в менее энергозависимой части иерархической памяти. Определенные изменения в объектах определенных типов понуждают систему автоматически выполнять функции этой команды. Система располагает также несколькими командами с необязательными параметрами, задающими операцию *принудительная запись*. Эти команды обуславливают выполнение системой так называемой *сквозной записи*. Цель последней — сохранение копии объекта в иерархической памяти.

4. При формировании объекта создающая его программа может задавать «класс функционирования» этого объекта. Она может указывать объем информации объекта, подлежащий передаче в основную память при обращении к этому объекту (обычно это секция размером 512 или 4096 байт).

5. МІ-интерфейс содержит команды пересылки объектов на временное хранение в перемещаемую запоминающую среду (гибкие магнитные диски или магнитная лента), не являющуюся составной частью одноуровневой памяти, а также команды

загрузки объектов из этой среды в одноуровневую память.

6. Под М1-интерфейсом расположены значительные по своим возможностям средства программного обеспечения, предназначенные для восстановления объекта после аномального завершения работы системы. При повторном пуске системы создается *список восстановления*, указывающий поврежденные объекты или те объекты, которые могли быть повреждены. Если объект определен как поврежденный, большинство операций с ним запрещено для сохранения целостности всей системы. М1-интерфейс содержит также команду RECLAIM, при выполнении которой освобождается любое пространство памяти, не связанное с действующим объектом, и производится возврат списка «утерянных» объектов, т. е. объектов, не принадлежащих никому.

7. М1-интерфейс содержит отдельный набор функций по выполнению операций ввода-вывода для устройств без памяти.

В IMP1-интерфейсе одноуровневая память реализуется с помощью принципов виртуальной и страничной организации памяти. IMP1-машина имеет традиционную архитектуру фон Неймана, хотя и с некоторыми нетрадиционными характеристиками. К последним следует отнести обширный диапазон адресов. Размер адреса равен 48 бит, что обеспечивает объем линейной виртуальной памяти, равный 2^{48} , или $281 \cdot 10^{12}$ байт.

Объем виртуальной памяти оказывается достаточно большим для того, чтобы включить в систему все запоминающие устройства. Виртуальная память размером 2^{48} байт с двухуровневой иерархией запоминающих устройств (память произвольного доступа и накопители на магнитных дисках) позволяет создать одноуровневую запоминающую среду, в которой перемещение данных осуществляется страницами фиксированного размера (512 байт). Виртуальный адрес 48 бит воспринимается машиной как состоящий из двух частей: 39-битового адреса страницы и 9-битового смещения внутри этой страницы. Аппаратные средства обработки адреса преобразуют первые 39 бит в адрес кадра 512-байтовой страницы в основной памяти (или генерируют сигнал прерывания при отсутствии страницы в основной памяти) и добавляют к нему 9-битовое смещение, формируя таким образом адрес области основной памяти. Из-за большого размера адреса вместо традиционных линейных таблиц страниц используются так называемые *хэш-таблицы*¹⁾.

Перемещение данных между отдельными устройствами двухуровневой иерархии запоминающих устройств осуществляется

¹ Речь идет о таблицах адресации данных, положение элемента в которых определяется преобразованием имени элемента данных в адрес фиксированной длины. — *Прим. ред.*

с помощью программных средств IMP1-архитектуры подобно тому, как выполняются операции над страницами в других системах. Объекты располагаются в сегментах 2^{48} -байтового виртуального адресного пространства, использующего для небольших объектов 64К-байтовые сегменты, а для больших объектов — 16М-байтовые. В состав программного обеспечения входит несколько справочников виртуальных адресов с соответствующими им адресами областей памяти на дисках. Для облегчения процесса восстановления (например, в случае повреждения указанных справочников) все страницы на магнитных дисках являются самоопределяемыми, поскольку содержат свой виртуальный адрес в качестве префикса.

Потенциальные адреса одноуровневой памяти в MI-интерфейсе имеют размер, равный 16 байт. Шесть байт такого адреса используются для записи виртуального адреса объекта, к которому имеет место потенциальная адресация.

Языками программирования Системы 38 являются РПГ¹⁾ и КОБОЛ. Использование принципа одноуровневой памяти не привело ни к каким изменениям этих языков. Следствием этого является то, что использование указанного принципа не имеет значения для прикладного программиста: прикладные программы выполняют операции ввода-вывода так же, как это имело бы место в других системах. Достоинства одноуровневой памяти не являются средствами, непосредственно предоставляемыми пользователям системы, хотя именно благодаря им конструкторы фирмы IBM сумели достичь значительных успехов в разработке программного обеспечения рассматриваемой системы.

УПРАВЛЕНИЕ ПРОЦЕССОМ

Как указано в гл. 2, существует большой семантический разрыв между современными операционными системами и архитектурой машины, обеспечивающих работу этих систем. Операционные системы представляют собой совокупность больших, чрезвычайно сложных программ, проектирование и разработка которых часто является «узким местом» проекта новой вычислительной системы, причем операционная система часто препятствует достижению требуемого быстродействия в современных системах.

Частичным решением этой проблемы является переложение некоторых традиционных функций операционной системы на

¹⁾ Сокращение, соответствующее английской аббревиатуре RPG (Report Program Generator). — *Прим. перев.*

аппаратные средства. В остальных случаях необходимо определять границу между тем, что можно вменить в обязанность машине, и тем, что необходимо сохранить как функции, реализуемые программными средствами. Первую категорию функций операционной системы можно назвать функциями, определяемыми «механизмами» системы, а вторую — функциями, определяемыми «политикой» выполнения заданий. Типичными функциями первой категории являются управление процессом, защита информации от несанкционированного доступа, управление памятью. Типичные функции второй категории — организация выполнения заданий, оценка стоимости выполнения заданий на ЭВМ, идентификация заданий директивами пользователя. Поскольку функции второй категории меняются с изменением внешних условий работы системы (часто эти изменения должен вносить системный программист) и базируются на возможностях упомянутых «механизмов» системы, представляется целесообразным осуществлять реализацию указанных «механизмов» аппаратными средствами и избегать подобной реализации функций, определяемых «политикой» выполнения задания пользователей.

Перечислим функции операционной системы, реализация которых с помощью аппаратных средств представляется наиболее вероятной:

1. *Управление процессом*: коммутация процессора (процессоров) между процессами; создание и уничтожение процессов; синхронизация процессов; организация связи между процессами.

2. *Управление памятью*: распределение пространства памяти; неявное перемещение информации между отдельными устройствами иерархической памяти (например, пересылка страниц); явное перемещение информации между ними (ввод-вывод файлов).

3. *Защита от несанкционированного доступа*: защита программ и данных друг от друга; передача средств защиты.

Поскольку управление памятью было освещено при рассмотрении средств функционирования подпрограмм и одноуровневой памяти, а защита от несанкционированного доступа — при анализе потенциальной адресации, в данном разделе главным образом рассматриваются вопросы управления процессом.

Процесс, или задача, — это последовательное выполнение потока команд, т. е. выполнение нескольких команд без совмещения во времени. Большинство современных операционных систем допускает совмещение во времени нескольких процессов. (Если в распоряжении системы находится только один процессор, то указанное совмещение является своего рода иллюзией, создаваемой за счет разделения процессов на временные

интервалы и перехода процессора от обслуживания некоторого интервала одного процесса к интервалу другого процесса.)

Рассмотрим основные функции операционной системы по управлению процессами.

Коммутация процессоров. Поскольку число процессов редко равно числу аппаратно реализованных процессоров и, как правило, нуждающихся в обслуживании процессов больше, необходим механизм распределения процессоров между процессами. Вычислительная система обычно предоставляет средства для переключения процессора на обслуживание другого процесса в следующих трех случаях: 1) когда текущий процесс достигает точки, в которой выполнение должно быть приостановлено (т. е. должно наступить «состояние ожидания»), 2) когда приостановленный процесс более высокого приоритета выходит из состояния ожидания или 3) когда текущий процесс выполняется в течение слишком долгого времени (прекращение непрерывного обслуживания такого процесса осуществляется для обеспечения «продвижения вперед» всех процессов или по крайней мере для создания видимости равного продвижения всех процессов).

Механизму, переключающему процессор на другой процесс, необходимо знать, какому из других процессов отдать предпочтение. Подобные решения часто базируются на стратегии распределения приоритетов (каждому процессу присваивается определенное значение приоритета: наивысший приоритет получает процесс, которому отдается наибольшее предпочтение) или стратегии «предельного срока». Во втором случае каждому процессу назначается предельный срок его обслуживания процессором. Возврат к продолжению незавершенного процесса осуществляется согласно расписанию, при этом в памяти системы хранятся предельные сроки однократного обслуживания всех процессов. Поскольку указанные стратегии относятся к так называемой «политике» выполнения заданий и часто подвергаются модификации при настройке системы на конкретное применение, следует избегать реализации конкретной стратегии в архитектуре машины; однако в ней должны быть предусмотрены определенные механизмы, обеспечивающие воплощение этих стратегий программными средствами.

Создание процесса. Существует немного практических ситуаций, в которых приемлемо использование вычислительной системы с фиксированным набором процессов. Чаще всего возникает потребность в механизме, позволяющем создавать одни процессы и уничтожать другие, пересылать данные и (или) потенциальные адреса формируемым процессам.

Управление одного процесса другим. Система должна располагать средствами, позволяющими одному процессу управ-

лять протеканием другого процесса. Типичные средства такого рода получили название SUSPEND (прекращение выполнения другого процесса без его разрушения) и RESUME (разрешение продолжения выполнения приостановленного процесса). Указанные средства оказываются полезными при реализации программными средствами стратегий обслуживания процессов по расписанию.

Синхронизация и связь. Эти разновидности управления процессами рассматриваются вместе, поскольку иногда для их реализации используется один и тот же механизм. Синхронизация необходима, когда несколько процессов одновременно нуждается в том или ином ресурсе, допускающем только последовательное использование, т. е. использование в порядке очереди. Средства межпроцессорной связи необходимы для передачи информации (данных, синхросигналов и т. п.) от одного процесса к другому. Имеется обширная литература по операционным системам, в которой освещаются принципы построения соответствующих механизмов, таких, как семафоры, почтовые ящики, средства приема и передачи, критические секции, мониторы, очереди и т. п. [37—43].

Управление ресурсами. Система нуждается также в средствах ограничения ресурсов (например, пространства памяти), которые могут использоваться процессом.

УПРАВЛЕНИЕ ПРОЦЕССАМИ В АРХИТЕКТУРЕ МАШИНЫ SWARD

Обратимся к архитектуре машины SWARD с целью рассмотрения конкретного примера аппаратных средств «поддержки» принципов управления процессом. Ранее упоминалось, что для этой архитектуры допускается использование объектов пяти типов, два из которых имеют прямое отношение к управлению процессами.

Речь идет о следующих объектах:

1) *модуль* — набор ячеек данных, определяющих область (домейн) санкционированного доступа, и последовательность машинных команд; последние могут представлять собой одну процедуру, набор процедур или процедуру с множеством точек входа;

2) *память данных* — динамически выделяемая память для ячеек данных;

3) *запись активации* — набор ячеек данных, представляющих ту часть домена адресации модуля, которая динамически выделяется при каждом переходе модуля в активное состояние (активация модуля); при переходе модуля в это состояние создается одна запись активации;

4) *процесс-машина* — объект, выполняющий параллельно с другими процесс-машинами один или несколько модулей;

5) *порт* — объект, через который один процесс может посылать значения данных в другой процесс.

Процесс-машина представляет собой процессор, который выполняет последовательную программу. Степень взаимосвязи этого процессора с аппаратно реализуемыми процессорами определяется конкретным техническим решением проекта вычислительной системы. Эта взаимосвязь скрыта в архитектуре системы, т. е. программы остаются в «неведении» относительно того, разделяют ли между собой несколько процесс-машин один реальный процессор или m процесс-машин — n реальных процессоров. Принципиально процесс-машина выполняет роль исполнителя команд, счетчика команд, стека записей активации и другой информации о состоянии системы, хотя ее архитектура определяется только командами, которые могут обращаться к этой процесс-машине.

Процессы создаются путем выполнения команды **CREATE-PROCESS-MACHINE**, схожей с командой **CALL**, но содержащей дополнительный операнд, в который осуществляется возврат адреса создаваемой процесс-машины. Иначе говоря, эта команда означает: «Начать выполнение указанного модуля с заданной точки входа как параллельного процесса и произвести возврат потенциального адреса созданной процесс-машине».

Процесс-машины уничтожаются, во-первых, путем выполнения инвариантной к типу обрабатываемых данных команды **DESTROY**, в качестве операнда которой задается потенциальный адрес данной процесс-машины; во-вторых, выполнением процесс-машиной команды **RETURN** при начальной активации этого процесса; в-третьих, если эта возможность указана необязательным параметром команды **CREATE-PROCESS-MACHINE**, при разрушении процесс-машины, выполняющей процесс, который создан этой процесс-машиной. Любая процесс-машина, располагающая потенциальным адресом другой процесс-машины, может задерживать выполнение операций в последней процесс-машине, возобновить их, изменить приоритет их выполнения или сформировать признак наличия ошибки посредством команды **CONTROL-PROCESS-MACHINE**.

Две машинные команды, **SEND** и **RECEIVE**, а также объект «порт» являются средствами коммуникации между несколькими процессами. Команда **SEND** определяется почти так же, как и команда **CALL**. Исключением является содержание пересылаемой информации и ее адресат: команда **CALL** передает управление и набор фактических параметров в точку входа указанного модуля, а команда **SEND** посылает набор значений параметров через указанный порт. Команда **RECEIVE** пересыла-

ет набор значений из порта в группу указанных локальных ячеек.

Команды SEND и RECEIVE выполняются синхронно. Когда, например, процесс-машина А выполняет команду, предписывающую пересылку значений содержимого ячеек X и Y через порт K, выполнение процесса приостанавливается, если другая процесс-машина не ожидает приема у этого порта. Готовность к приему может возникнуть у нее только как результат выполнения команды RECEIVE. Как только это происходит, возобновляется выполнение приостановленного процесса. Взаимодействие этих команд подобно так называемому *свиданию процессов* (rendezvous) в языке Ада [44]. В результате вместо очереди сообщений (наборов значений переменных) у порта имеет место очередь процесс-машин.

Описанный механизм функционирования интерфейса приемо-передачи служит также для выполнения проверки данных на соответствие типов. Это возможно благодаря тому, что в рассматриваемой архитектуре используется теговая память.

Механизм приемо-передачи можно использовать и для защиты так называемой *критической секции* программы от одновременного выполнения ее команд многопроцессорными машинами, хотя его использование для этой цели может оказаться неудобным и неэффективным. Однако в данной архитектуре реализован другой, чрезвычайно простой механизм синхронизации, который базируется на принципе синхронизации последовательностей команд, лежащем в основе монитора — этого важного элемента программного обеспечения вычислительных систем [37].

Монитор — это совокупность одной или нескольких процедур, использующих определенные ресурсы системы, например таблицу данных, буферный пул, устройство ввода-вывода. Монитор имеет двойное назначение: во-первых, сокрытие структуры и физических характеристик ресурсов с целью улучшения структуры системы в целом; во-вторых, выполнение необходимой синхронизации при пользовании ресурсами (только одному процессу в каждый момент времени разрешается выполнение всех или части машинных кодов, содержащихся в мониторе). Вместо того чтобы предоставлять всем программам непосредственный обзор ресурсов и необходимые средства синхронизации пользования ими, архитектура системы предполагает следующее: для выполнения операции с ресурсами каждый процесс обращается к монитору. Отметим, что изучение одновременного обслуживания процессов средствами языка Ада, обеспечивающего необходимые коммуникации между процессами, но не располагающего мониторами, показывает необходимость обоих механизмов [59].

Поскольку в программе, принадлежащей более абстрактному описанию машины SWARD, чем ее архитектура, монитор представляется как объект «модуль», в архитектуре SWARD имеется простой механизм синхронизации, ориентированный на модули. Он определяется командами GUARD и UNGUARD, которые не имеют операндов. Когда процесс-машина выполняет содержащуюся в модуле команду GUARD, происходит одно из двух действий. Если модуль не находится в состоянии «охраняем» (guarded state), он переходит в это состояние, и выполнение продолжается со следующей команды. Если же модуль уже пребывает в указанном состоянии, выполнение действий процесс-машины приостанавливается на этой команде, и процесс-машина помещается в очередь на данный модуль в состоянии «охраняем». Выполнение команды UNGUARD заставляет модуль выйти из состояния «охраняем», после чего выполнение продолжается со следующей команды. Если какая-нибудь процесс-машина находится в очереди на данный модуль в состоянии «охраняем», то первая из таких процесс-машин из очереди удаляется, а ее операции возобновляются.

Принцип использования процесс-машин «ортогонален» всем другим принципам рассматриваемой архитектуры, таким, как адресация и распределение памяти. Этот принцип базируется на модели архитектуры, в распоряжении которой имеется неограниченное количество процессоров. Однако принцип одноуровневой памяти предполагает использование пусть большого, но фиксированного числа запоминающих устройств. Следовательно, возникает потребность в механизме, ограничивающем объем памяти, который может быть запрошен процесс-машиной как для управления системой, так и для предотвращения создания процессом объектов, «ускользающих» из-под его контроля. Работа такого механизма основывается на использовании двух количественных характеристик объема памяти (amount of memory): *АМ* и *АМ имеющейся в распоряжении памяти*.

Будучи атрибутом объекта, АМ является в некотором смысле расплывчатой мерой памяти, занимаемой объектом. Поскольку объекты непосредственно программам не доступны («не видны») и в связи с этим могут менять свою форму при переходе от одной реализации данной архитектуры к другой, АМ не является мерой точного числа байтов или битов, занимаемых объектом. Это только некоторая грубая относительная оценка указанного параметра. Архитектура машины позволяет дать, например, следующие оценки:

порт занимает приблизительно 0,005—0,01 АМ;

объект «память данных», предоставляющий массив двухсот 10-символьных полей, занимает ~1 АМ;

объект «процесс-машина» занимает примерно 0,1—0,3 АМ.

При задании характеристик архитектуры указывается, что программы не должны использовать в качестве точных значений величины, измеряемые в АМ. (Отметим, что при реализации прототипа машины SWARD АМ считается равным 16К бит).

Помимо того что процесс-машина сама занимает некоторую память размером АМ, в ее распоряжении для создания объектов имеется еще некоторый объем свободной памяти, оцениваемый характеристикой «АМ имеющейся в распоряжении памяти». Измеряемая в АМ память, необходимая для создания объекта, заимствуется из памяти, имеющейся в распоряжении данной процесс-машины. Принято говорить, что «АМ создаваемого объекта вычитается из АМ имеющейся в распоряжении памяти». Количество АМ имеющейся в распоряжении памяти, которое подлежит пересылке от создающей процесс-машины к создаваемой процесс-машине, задается в виде операнда команды CREATE-PROCESS-MACHINE. Кроме того, имеется команда TRANSFER-AM, позволяющая одной процесс-машине (если она владеет необходимыми потенциальными адресами) перемещать между двумя другими процесс-машинами АМ имеющейся в распоряжении памяти.

При рассмотрении организации одноуровневой памяти отмечалось, что для представления устройств ввода-вывода без памяти нужны специальные средства. В архитектуре SWARD для этих целей используются некоторые средства управления процессом. Принята следующая модель: устройства без памяти (терминалы, устройства печати и т. п.) представлены потенциальными адресами, а для выполнения операций ввода-вывода используются команды SEND и RECEIVE. Терминал можно рассматривать как порт, а работающего за терминалом человека-оператора — как процесс. Такая модель имеет два достоинства. Во-первых, она позволяет пользователю заменять процессы устройствами ввода-вывода и, наоборот, замещать эти устройства процессами без изменения программы. Во-вторых, благодаря синхронному режиму работы механизма приема-передачи устраняется потребность в использовании системы прерываний, что позволяет архитектуре машины придерживаться единообразного принципа асинхронной работы, реализуемого в виде процесс-машины. (Асинхронные операции устройств ввода-вывода без памяти можно реализовать путем создания отдельного процесса для их выполнения.)

При попытке реализации той или иной функции системы средствами, на которых базируется архитектура системы, возникают проблемы функционирования и отладки соответствующих программ, для которых (и тем самым для людей, их создающих) становится «невидимой» информация о состоянии системы. В машине SWARD для решения этих проблем вводится

дополнительно несколько команд. В одной из них — команде DESCRIBE-CAPABILITY — в качестве операндов используются ячейка указателя и массив. Эта команда заполняет массив информацией о задаваемом в указателе потенциальном адресе и объекте (или его составной части), к которому этот потенциальный адрес относится. Информация о потенциальном адресе включает директивную информацию о так называемых *правах доступа*, которыми этот адрес «владеет». Информация об объекте зависит от типа последнего. Если, например, это — объект «модуль», рассматриваемая команда дает следующую информацию: 1) каков объем (в единицах, называемых АМ) памяти, занимаемой объектом; 2) активен ли объект (т. е. имеет ли запись активации); 3) находится ли объект в состоянии «охраняем»; 4) подлежит ли модуль уничтожению явным образом при уничтожении создающей процесс-машины. Для объекта «порт» команда DESCRIBE-CAPABILITY указывает количество процесс-машин, стоящих в очереди к этому порту. Для объекта «процесс-машина» данная команда указывает в единицах АМ объем памяти, предоставленный в распоряжение этому объекту, и состояние последнего: активен ли он, приостановлено ли его выполнение, блокирован ли он модулем, находящимся в состоянии «охраняем», ожидает ли своей очереди у порта.

Вторая команда DESCRIBE-PROCESS-STACK предназначена именно для процесс-машин. Располагая потенциальным адресом для процесс-машины, эта команда возвращает в операнд, являющийся массивом, описание текущего стека активации процесс-машины. Каждый элемент описывает активацию модуля, поскольку содержит потенциальный адрес модуля (потенциальный адрес не располагает директивной информацией с правом создания защиты от несанкционированного доступа) и индекс следующей команды, подлежащей выполнению.

ДРУГИЕ ПРИМЕРЫ

Помимо рассмотренного выше известны и другие предложения по коммутации процессов и их синхронизации средствами машины [12, 45, 46]. Примером промышленно изготавливаемых машин такого типа являются ЭВМ фирмы Honeywell, серия 60, уровень 64 [47, 48]. Для того чтобы машина могла идентифицировать процессы, строится блок управления процессами и выполняется команда «Начало процесса». Коммутацию процессов осуществляет машина, используя блок управления процессами для записи в него на хранение информации о состоянии системы. Для синхронизации процессов набор команд включает так называемый механизм типа *семафор*.

В микропроцессоре Intel 432 большая часть функций по управлению процессами выполняется непосредственно кристаллом благодаря использованию того же принципа, что и в машине SWARD (разделение указанных функций на определяемые «механизмами» системы и определяемые «политикой» выполнения заданий), при этом определяется похожий набор объектов. Более подробно этот вопрос рассматривается в гл. 17.

ДОСТОИНСТВА УПРАВЛЕНИЯ ПРОЦЕССАМИ

Повышенное быстродействие системы. Возложение на машину функций создания процессов, их коммутации и синхронизации оказывает значительное влияние на скорость выполнения этих функций, что в свою очередь приводит к увеличению эффективности выполнения программ и улучшению стиля программирования. Многие специалисты признают, что использование одновременно протекающих процессов является естественным и рациональным способом структурирования многих прикладных задач. Однако программисты часто избегают таких процессов, поскольку при этом повышаются накладные расходы (за счет частого создания и уничтожения процессов). Передача функций управления процессами машине помогает преодолеть этот нежелательный эффект путем снижения не только абсолютных, но и относительных величин накладных расходов по созданию процессов по сравнению с другими операциями, например операцией сложения. Так, в машине SWARD команда CREATE-PROCESS-MACHINE является заменителем функции команды CALL, но в аппаратно реализованном прототипе машины время ее выполнения только на 35% больше времени выполнения команды CALL.

Расширенные возможности реализации системы. Как и при анализе эффекта использования других принципов организации вычислительной системы, рассмотренных в этой главе, включение управления процессами в состав функций аппаратных средств машины создает новые возможности для ее реализации. Для современных вычислительных систем с архитектурой низкого уровня организации основным средством повышения быстродействия является конвейерная обработка. Однако широкое применение конвейерной обработки, хотя и обеспечивает высокую эффективность работы вычислительной системы, обходится дорого вследствие необходимости выявлять взаимозависимость команд и из-за сложности алгоритмов обработки прерываний в последовательном потоке команд (содержащем, например, условные переходы и прерывания).

Введение в архитектуру системы таких простых средств управления процессами, какие предусмотрены в машине SWARD,

предоставляет разработчику вычислительной системы дополнительные возможности. Например, конструктор может строить машину, на которой базируется архитектура системы, в виде набора простых процессоров неконвейерного типа. Каждый процессор после потери возможности продолжать действовать как конкретная процесс-машина ищет другую процесс-машину с целью «подражания» ей. Но поскольку коммутация процессов осуществляется за интерфейсом машины (т. е. на аппаратном уровне), указанное выше при желании может осуществляться параллельно. Пока выполняются один или несколько процессов, схемы параллельной логики могут проводить анализ других существующих процессов с целью определения того, когда следует осуществлять коммутацию процессов и к какому процессу подключаться. Оснащение архитектуры сложными средствами синхронизации тоже предоставляет возможность выполнять эти операции (например, постановку в очередь или снятие с очереди) параллельно с другой работой.

Улучшенная структура операционной системы. Материал, изложенный в данном и нескольких предыдущих разделах, свидетельствует о тенденции к реализации некоторых базовых функций операционной системы непосредственно в машине, т. е. аппаратными средствами. Это позволяет дать таким функциям более стабильную основу и формализованное определение. Архитектура машины обычно лучше документирована и более тщательно определена, чем базирующееся на ней программное обеспечение. В результате операционная система становится менее сложной. Следует также учесть, что проектирование всех аппаратно реализуемых механизмов сопровождается значительно более тщательным тестированием их работоспособности. Кроме того, аппаратная реализация функций программного обеспечения во многом способствует устранению побочных эффектов этих функций, а также включению в программное обеспечение так называемых неортодоксальных интерфейсов. Указанные интерфейсы являются отклонениями от «стандарта» операционных систем и источником дополнительных проблем современного системного программирования. Все это позволяет с большим доверием относиться к утверждениям о корректности работы вычислительной системы в целом.

ТИПЫ АДРЕСАЦИИ

Важной характеристикой архитектуры вычислительной машины являются *типы адресации*, используемые набором ее команд, т. е. адресация команд к своим операндам. Эта характеристика связана с проблемами, присущими не только последним достижениям в архитектуре ЭВМ, но и традиционным решениям;

речь идет об относительных достоинствах различных типов адресации, представляемых моделями, ориентированными на использование стеков, регистров или памяти. Обсуждение достоинств и недостатков этих моделей [49—53] свидетельствует о том, что упомянутые проблемы не нашли еще своего окончательного решения.

Выбор типа адресации команд связан с поиском наиболее эффективных способов представления (в пространстве) и обработки (во времени) выражений. Важность этой проблемы не подлежит сомнению, так как вычисление значения выражения обычно является функцией, наиболее часто выполняемой вычислительной машиной. Например, вычисления выражений требуются следующим операторам языков высокого уровня:

всем операторам присваивания (например, $A=B$; $A=A+B$);
всем операторам IF (таким, как $IF(A=B)...$; $IF(A+B=C)...$);

многим операторам DO ($DO\ WHILE\ (A \leq B)$);

некоторым другим операторам (например, $CALL\ XYZ(Q, A+B)$).

Согласно результатам одного исследования [54], операторами присваивания и IF являются 78% всех выполняемых операторов, в другом исследовании [55] этот параметр указывается равным 59%. Следовательно, эффективность вычисления выражений в значительной степени определяет эффективность использования пространства памяти и машинного времени вычислительной системы.

Поскольку существует несколько основных типов адресации команд, каждый из них будет рассмотрен отдельно, а затем будет проведено их количественное и качественное сравнение. Отметим, что рассматриваемые здесь основные типы адресации не исчерпывают всех возможных способов адресации, кроме того, иногда применяют их сочетания. Примером могут служить команды микропроцессора iAPX 432, описываемого в части VI данной книги.

АДРЕСАЦИЯ ПОСРЕДСТВОМ РЕГИСТРОВ

Архитектура вычислительной машины, использующей для адресации набор регистров общего назначения, лучше всего известна программистам. Здесь команды получают возможность адресоваться к операндам, расположенным в одной из двух запоминающих сред, — основной памяти или регистрах. Размер регистра обычно фиксирован и совпадает с размером слова физически реализованной основной памяти. К любому регистру можно адресоваться непосредственно, поскольку регистры представлены в виде массива запоминающих элементов. Количество

регистров определяется архитектурой машины, а не конкретным вариантом ее реализации. Обычно регистров немного — 8 или 16. Технологическая база регистров обеспечивает более высокое быстродействие последних по сравнению с основной памятью.

Типичным является выполнение арифметических операций только в регистре; при этом команда содержит два операнда, размещаемые либо в регистрах, либо один из них находится в регистре, а другой — в основной памяти. Такие команды имеют следующие форматы:

КОП РЕГ ПАМ
КОП РЕГ РЕГ

где КОП — код операции, РЕГ — номер регистра, ПАМ — адрес основной памяти.

Не останавливаясь подробно на формате команд, отметим только, что ПАМ — это или линейный адрес памяти, или линейное смещение области памяти относительно адреса, указываемого специальным (базовым) регистром, или адрес ячейки домена санкционированного доступа и т. д. Будем полагать, что код операции помимо задания выполняемой операции указывает также, какой из двух форматов используется.

В качестве примера рассмотрим команды, генерируемые для оператора присваивания $A := B + C$

LOAD REG1,B
ADD REG1,C
STORE REG1,A

Для проведения в дальнейшем сравнения типов адресации введем следующие условные обозначения: Р — размер кода операции команд некой гипотетической архитектуры, использующей адресацию посредством регистров; R — размер адреса регистра; S — размер адреса операнда, расположенного в основной памяти; D — средний размер операнда.

АДРЕСАЦИЯ ПОСРЕДСТВОМ АККУМУЛЯТОРА

Данный тип адресации подразумевает ориентацию архитектуры машины на использование для адресации единственного регистра, называемого *аккумулятором*. В сущности, такая модель архитектуры является предельным вариантом архитектуры с адресацией посредством регистров, когда количество последних сведено к одному регистру — аккумулятору. Он играет роль неявно заданного операнда всех команд, имеющих одинаковый и единственный формат следующего вида:

КОП ПАМ

В данном случае для оператора $A := B + C$ генерируются следующие машинные команды:

```
LOAD B
ADD C
STORE A
```

При адресации посредством аккумулятора размер кода операции команды не совпадает с размером кода операции команды при адресации посредством регистров, поскольку в последнем случае код операции включает еще информацию о том, какой из двух возможных форматов команд используется. Архитектура машины с адресацией посредством аккумулятора позволяет ограничиться меньшим набором команд, так как при этом отсутствуют команды формата «регистр-регистр». Поскольку в проводимых здесь количественных оценках внимание уделяется только командам, используемым для вычисления значения выражений (т. е. команды, относящиеся к управлению функционированием системы, игнорируются), размер кода операции команды с адресацией посредством аккумулятора может быть определен как $P-1$, где P — код операции команды машины с адресацией посредством регистров общего назначения. Следовательно, в данном случае код операции может быть короче на 1 бит. Если же предположить, что независимо от типа адресации длина команды должна равняться одной и той же величине, то можно сказать, что в распоряжение архитектуры с адресацией посредством аккумулятора поступает дополнительно 1 бит (от каждой команды).

АДРЕСАЦИЯ ПОСРЕДСТВОМ СТЕКА С ИСПОЛЬЗОВАНИЕМ БЕЗАДРЕСНЫХ КОМАНД

Адресация этого типа основана на использовании не аккумулятора или набора регистров, а стека. (Рассматриваемое здесь применение стека не обязательно должно быть подобно использованию стеков рассмотренным ранее механизмом управления подпрограммами.) В общем случае команды неявно адресуются к элементу стека, расположенному на его вершине, или к двум верхним элементам стека. Такие команды могут иметь один из следующих форматов:

```
КОП ПАМ
КОП
```

К командам первого формата относятся команды **PUSH** и **STORE**. По команде **PUSH** из области основной памяти, адресуемой ее операндом, извлекается копия содержимого и загружается в стек; при этом на одну позицию вниз сдвигаются все

остальные элементы стека. По команде STORE из стека выталкивается элемент, расположенный на его вершине, и помещается в область памяти, адресуемую операндом этой команды.

Команды второго формата являются безадресными и имеют только код операции. Обычно они неявным образом адресуются к одному или двум элементам вершины стека. Типичными командами этой группы являются следующие:

ADD — сложение значений двух верхних элементов стека, удаление их из стека с последующей записью в него полученной суммы;

COMP — изменение на обратный знака элемента на вершине стека;

EQUAL — сравнение значений двух верхних элементов стека, извлечение их из стека и запись в него логической величины **TRUE** или **FALSE** в зависимости от результата сравнения;

POP — извлечение («выталкивание») из стека элемента, расположенного на его вершине.

Последовательность машинных команд, генерируемых для оператора $A := B + C$, в этом случае имеет следующий вид:

```
PUSH B
PUSH C
ADD
STORE A
```

Машина с подобной адресацией располагает примерно таким же количеством различных команд, как и при использовании адресации посредством аккумулятора, но их меньше, чем имеет машина с адресацией посредством регистров общего назначения. Следовательно, в данном случае размер кода операции тоже равен $\sim (P-1)$.

Прежде чем сравнивать рассматриваемый тип адресации с другими типами, следует проанализировать вопрос о реализации стека на практике. Теоретически часто полагают, что объем стека безграничен. Однако это не соответствует реальным возможностям аппаратных средств ЭВМ. Поэтому обычно предлагают следующие компромиссные решения:

1) выполнить стек в виде конечного, фиксированного набора регистров, расположенных в процессоре;

2) выполнить стек в виде конечного, фиксированного набора регистров, расположенных в процессоре, используя область основной памяти как вспомогательное средство (буфер) в случае переполнения набора регистров;

3) выполнить стек в виде области памяти. (Примером может служить архитектура машин, обсуждаемая в частях II и III данной книги.)

Практическое осуществление того или иного предложения отражается на пропускной способности тракта «процессор — память» и поэтому подлежит отдельному изучению. Первое из предлагаемых решений назовем *стеком на регистрах*, последнее — *стеком в памяти*. Они будут описаны отдельно. Второе предлагаемое решение может быть рассмотрено как некоторая аппроксимация первого.

АДРЕСАЦИЯ ПОСРЕДСТВОМ СТЕКА С ИСПОЛЬЗОВАНИЕМ ОДНОАДРЕСНЫХ КОМАНД

Адресация этого типа подобна последней из рассмотренных выше, за исключением следующей особенности: многие безадресные команды (состоящие только из кода операции) имеют одноадресный эквивалент. Например, формат такой команды ADD представляется в виде

ADD ПАМ

При выполнении этой команды содержимое области памяти, адресуемой операндом ПАМ, добавляется к значению элемента на вершине стека с последующим замещением верхнего элемента полученной суммой. Команды этого типа, генерируемые для оператора $A := B + C$, имеют следующий вид:

```
PUSH B
ADD C
STORE A
```

Поскольку для адресации данного типа желательно сохранение безадресных команд, наличие двух возможных форматов команд обуславливает размер их кода операции, равный Р.

Итак, при рассмотрении проблемы реализации стека возникают две задачи: выполнение стека на регистрах и в памяти.

АДРЕСАЦИЯ ТИПА «ПАМЯТЬ-ПАМЯТЬ»

Адресация данного типа не предполагает в архитектуре машины явного определения аккумулятора, регистров или стека; все операнды команд адресуются к областям основной памяти. Такие команды в большинстве случаев бывают двух- или трехадресными. Формат двухадресных команд имеет следующий вид:

КОП ПАМ ПАМ

Отметим, что адресация этого типа может быть дополнена небольшим количеством одноадресных команд, например для вычисления дополнения. Количество различных двухадресных команд, необходимых для реализации архитектуры машины с адресацией типа «память-память», очевидно, несколько меньше

количества необходимых команд при адресации посредством стека: в первом случае не нужны команды записи в стек и извлечения из него. Однако указанная разница незначительна, и поэтому размер кода операции двухадресных команд может быть оценен величиной $P-1$.

Другой возможный вид команд с адресацией типа «память-память» имеет формат

КОП ПАМ ПАМ ПАМ

Команды такого формата называют трехадресными, причем два операнда, как правило, являются «источниками» исходных данных, а третий — «приемником» результата. Архитектура машины с набором подобных трехадресных команд имеет код операции, размер которого равен $\sim (P-1)$.

Существует еще одна разновидность архитектуры ЭВМ, основанная на использовании адресации типа «память-память». Набор ее команд включает как двух-, так и трехадресные команды, а сама архитектура условно называется $2/3$ -адресной типа «память-память». Поскольку в ней используются два типа форматов команд (двух- и трехадресных), коду операции команды необходим дополнительный бит для указания типа ее формата, а поэтому размер кода операции можно принять равным $\sim P$.

КОЛИЧЕСТВЕННЫЙ АНАЛИЗ

Характеристики девяти типов адресации команд, подлежащие анализу, приведены в табл. 4.2. Следует обратить внимание на то, что некоторые характеристики, в частности объем пересылаемых данных, приходящийся на одну команду, меняются в зависимости от формата команд.

При проведении анализа будем рассматривать девять вариантов архитектуры гипотетической вычислительной машины, идентичных во всем, за исключением типа адресации, используемого командами машины. Упомянутая идентичность подразумевает наличие у машин одинакового набора команд (за исключением тех команд, в которых применяется специфический для данной архитектуры способ адресации, например команды LOAD, PUSH) и эквивалентных размеров S адресов операндов в основной памяти.

Поскольку эффективность того или иного типа адресации, используемого командами, как будет показано, зависит от вида, размера и сложности вычисляемых выражений, анализ должен проводиться с привлечением выражений с различными характеристиками, а также с учетом частоты, с которой такие выражения встречаются.

Таблица 4.2. Характеристики типов адресации команд

Тип адресации	Количество различных форматов команды	Длина команды, бит	Объем пересылаемых данных на одну команду ¹⁾
Посредством регистра (РЕГ)	2	$P + R + S$ $P + 2R$	D 0
Посредством аккумулятора (АКК)	1	$P - 1 + S$	D
Посредством стека на регистрах и безадресных команд (СРБА)	2	$P - 1 + S$ $P - 1$	D 0
Посредством стека в памяти и безадресных команд (СПБА)	2	$P - 1 + S$ $P - 1$	2D 2D или 3D ²⁾
Посредством стека на регистрах и одноадресных команд (СРОА)	2	$P + S$ P	D 0
Посредством стека в памяти и одноадресных команд (СПОА)	2	$P + S$ P	2D или 3D ²⁾ 2D или 3D ²⁾
«Память-память» с использованием двухадресных команд (ППДА)	1	$P - 1 + 2S$	2D или 3D ²⁾
«Память-память» с использованием трехадресных команд (ППТА)	1	$P - 1 + 3S$	2D или 3D ²⁾
«Память-память» с использованием двух- и трехадресных команд (ППДТА)	2	$P + 2S$ $P + 3S$	2D или 3D ²⁾ 2D или 3D ²⁾

¹⁾ Подсчитываются только пересылки данных в памяти (исключаются пересылки между регистрами и подобные операции с аккумулятором и стеком).

²⁾ Для таких команд, как PUSH и MOVE, объем пересылаемых данных составляет 2D, а для команд, подобных команде ADD, равен 3D.

Начнем анализ, взяв за основу для сравнения следующие операторы присваивания (выбор этих операторов из почти бесконечного набора возможных операторов станет понятным позже):

1) $A := B$; 2) $A := A + B$; 3) $A := B + C$; 4) $A := B + C + D$;
5) $A := (B + C) * (D + E)$; 6) $A := B + C + D + E$; 7) $A := (B * C + D * E) * F + G$;

Оставим пока на время вопрос о частоте выполнения этих выражений. В табл. 4.3 показано количество команд, необходимых для выполнения операторов машинной с архитектурой того или иного вида. Две разновидности адресации посредством стека в памяти не рассматриваются, поскольку от адресации посредством стека в регистрах они отличаются только пересылкой данных из памяти в память. Отметим, что при адресации ти-

Таблица 4.3. Количество команд, необходимых для выполнения операторов машинами с разными видами архитектуры

Номер оператора	РЕГ	АКК	СРБА	СРОА	ППДА	ППТА	ППДТА
1	2	2	2	2	1	1	1
2	3	3	4	3	1	1	1
3	3	3	4	3	2	1	1
4	4	4	6	4	3	2	2
5	6	7	8	6	5	3	3
6	5	5	8	5	4	3	3
7	8	9	12	8	7	5	5

па «память-память» во всех случаях требуется меньшее количество команд.

В табл. 4.4 в символической форме представлены размеры генерируемых команд: размер каждой команды указан в виде алгебраического выражения, составленного из переменных P (размер кода операции в соответствии с конкретным вариантом архитектуры машины), R (размер адреса регистра) и

Таблица 4.4. Размеры команд для машин различной архитектуры

Номер оператора	Размер команды, бит			
	РЕГ	АКК	СРБА	СРОА
1	$2P + 2R + 2S$	$2P + 2S - 2$	$2P + 2S - 2$	$2P + 2S$
2	$3P + 3R + 3S$	$3P + 3S - 3$	$4P + 3S - 4$	$3P + 3S$
3	$3P + 3R + 3S$	$3P + 3S - 3$	$4P + 3S - 4$	$3P + 3S$
4	$3P + 3R + 3S$	$4P + 4S - 4$	$6P + 4S - 6$	$4P + 4S$
5	$6P + 7R + 5S$	$7P + 7S - 7$	$8P + 5S - 8$	$6P + 5S$
6	$5P + 5R + 5S$	$5P + 5S - 5$	$8P + 5S - 5$	$5P + 5S$
7	$8P + 9R + 7S$	$9P + 9S - 9$	$12P + 7S - 12$	$8P + 7S$

Номер оператора	Размеры команды, бит		
	ППДА	ППТА	ППДТА
1	$P + 2S - 1$	$P + 2S - 1$	$P + 2S$
2	$P + 2S - 1$	$P + 3S - 1$	$P + 2S$
3	$2P + 4S - 2$	$P + 3S - 1$	$P + 3S$
4	$3P + 6S - 3$	$2P + 6S - 2$	$2P + 5S$
5	$5P + 10S - 5$	$3P + 9S - 3$	$3P + 8S$
6	$4P + 8S - 4$	$3P + 9S - 3$	$3P + 7S$
7	$7P + 14S - 7$	$5P + 15S - 5$	$5P + 12S$

Таблица 4.5. Число операций ЧТЕНИЕ и ЗАПИСЬ для данных, размещаемых в памяти (пересылка данных длиной D)

Номер оператора	Число операций								
	РЕГ	АКК	СРВА	СПВА	СРОА	СПОА	ППДА	ППТА	ППДТА
1	2	2	2	4	2	4	2	2	2
2	3	3	3	9	3	7	3	3	3
3	3	3	3	9	3	7	5	3	3
4	4	4	4	14	4	10	8	6	6
5	5	7	5	19	5	15	13	9	9
6	5	5	5	19	5	13	11	9	9
7	7	9	7	29	7	21	19	15	15

S (размер адреса операнда, расположенного в основной памяти).

В табл. 4.5 приведены значения числа операций с данными, размещаемыми в основной памяти. В большинстве случаев каждое такое значение равно коэффициенту при переменной S в соответствующей позиции табл. 4.4. Различия наблюдаются для команд, использующих адресацию посредством стека в памяти или адресацию типа «память-память». Например, для выполнения команды ADD с адресацией посредством стека в памяти при безадресном формате необходимо три обращения к памяти. Такое же количество обращений к памяти требуется для выполнения двухадресной команды ADD с адресацией типа «память-память». Взятые вместе табл. 4.4 и 4.5 характеризуют использование тракта «процессор-память».

Теперь необходимо рассмотреть относительную частоту использования знаков операций в семи операторах, выбранных в качестве примера. В табл. 4.6 приведены усредненные статистические данные, полученные в результате проведенных исследований [54, 55]. Первая группа данных показывает число знаков операций, используемых справа от знака операции при

Таблица 4.6. Статистические данные, отображающие сложность оператора присваивания языка высокого уровня

Количество знаков операций	Относительная частота использования знаков операций, %	Знаки арифметических операций	Относительное количество знаков операций, %
0	72,1	Знаки одноместных операций	1,5
1	20,5	Знаки двухместных операций: + и —	84,0
2	3,8	Знаки двухместных операций: *, / и **	14,5
3	3,3		
4	0,3		

сваивания; вторая группа — значения относительной частоты использования знаков арифметических операций.

Согласно данным табл. 4.6, частота использования оператора 1 ($A:=B$) равна 72,1%. Совместная частота использования операторов 2 и 3 составляет 20,5%. Эти операторы объединены потому, что машины с различными типами адресации команд используют их с разной эффективностью. Грубые оценки [51] показывают, что частота выполнения оператора 2 равна 14,4%, а оператора 3—6,1%. У оператора 4 два знака операций, а поэтому частота его использования составляет 3,8%. Два оператора (5 и 6) имеют по три знака операции каждый. Однако один из них (оператор 5) требует соблюдения определенной последовательности выполнения операций, для другого (оператора 6) порядок реализации операций не имеет значения. Вот почему эти два оператора рассматриваются вместе. Совместная частота их использования, согласно табл. 4.6, равна 3,3%. Для определения частоты использования каждого из указанных операторов необходимо знать вероятность, с которой в выражении с тремя знаками операций встречаются операции умножения и деления или сложения и вычитания). Если воспользоваться данными табл. 4.6, то указанную величину можно определить следующим образом: $0,84^3 + 0,145^3 \approx 60\%$. Следовательно, частота использования операторов 5 и 6 равна соответственно 1,3 и 2,0%. Оператор 7, содержащий пять знаков операций, позволяет получить представление о всех прочих разновидностях оператора присваивания; частота его использования составляет 0,3%. Значения частоты использования всех перечисленных операторов сведены в табл. 4.7.

Таблица 4.7. Частота использования операторов присваивания различного вида

Оператор	Частота использования, %
$A:=B$;	72,1
$A:=A+B$;	14,4
$A:=B+C$;	6,1
$A:=B+C+D$;	3,8
$A:=(B+C)*(D+E)$;	1,3
$A:=B+C+D+E$;	2,0
$A:=(B*C+D*E)*F+G$;	0,3

На основании данных табл. 4.4 и 4.7 можно вывести ряд формул для оценки некоторых параметров вычислительных машин с адресацией различного типа, выполняющих операторы присваивания рассмотренного вида. Эти формулы сведены в табл. 4.8.

Табл. 4.9 содержит сравнительные характеристики различных типов адресации по нескольким параметрам. Представленные в таблице значения получены путем вычисления по формулам, приведенным в табл. 4.8, с последующей нормализацией

Таблица 4.8. Сравнение некоторых параметров выполнения операторов присваивания машинными с различными типами адресации

Тип адресации	Средний размер машинной команды, бит	Средний объем данных, пересылаемых между памятью и процессором, бит
Посредством регистров	$2,4P + 2,4R + 2,4S$	$2,4P + 2,4R + 2,4S + 2,4D$
Посредством аккумулятора	$2,4P + 2,4S - 2,4$	$2,4P + 2,4S - 2,4 + 2,4D$
Посредством стека на регистрах и безадресных команд	$2,8P + 2,4S - 2,8$	$2,8P + 2,4S - 2,8 - 2,4D$
Посредством стека в памяти и безадресных команд	$2,8P + 2,4S - 2,8$	$2,8P + 2,4S - 2,8 + 6,0D$
Посредством стека на регистрах и одноадресных команд	$2,4P + 2,4S$	$2,4P + 2,4S + 2,4D$
Посредством стека в памяти и одноадресных команд	$2,4P + 2,4S$	$2,4P + 2,4S + 5,2$
«Память-память» с использованием двухадресных команд	$1,3P + 2,5S - 1,3$	$1,3P + 2,5S - 1,3 + 2,9D$
«Память-память» с использованием трехадресных команд	$1,1P + 2,6S - 1,1$	$1,1P + 2,6S - 1,1 + 2,6D$
«Память-память» с использованием двух- и трехадресных команд	$1,1P + 2,4S$	$1,1P + 2,4S + 2,6D$

содержимого каждого столбца значений посредством приравнивания единице наименьшего (наилучшего) из них. С точки зрения компактности команд (параметр S) наилучшей оказывается адресация типа «память-память», а наихудшей — адресация посредством регистров общего назначения. Сравнение типов адресации по общему количеству пересылаемых битов между памятью и процессором (параметр M) не дает столь ясно выраженных результатов. Адресация типа «память-память» с использованием двух- и трехадресных команд оказывается наилучшей во всех трех случаях измерения параметра M , адресация посредством стека в памяти — наихудшей, позиции остальных типов адресации, классифицируемых по данному параметру, не являются столь очевидными.

Если воспользоваться теми же приближенными оценками, что и раньше, то можно считать несущественным 10%-ный разброс значений величин, содержащихся в таблице. Тогда оказывается, что типы адресации «память-память», посредством стека на регистрах безадресными и одноадресными командами, а также посредством аккумулятора оказываются наивысшего ранга по параметру M , несколько худшие результаты дает ад-

Таблица 4.9. Сравнение различных типов адресации по основным параметрам

Тип адресации	Нормализованный размер команды (параметр S)		Нормализованный объем данных, пересылаемых между процессором и памятью (параметр M)		
	P = 8 R = 4 S = 12	P = 8 R = 4 S = 16	D = 16 P = 8 R = 4 S = 12	D = 32 P = 8 R = 4 S = 12	D = 32 P = 8 R = 4 S = 16
Посредством регистров	1,53	1,42	1,21	1,11	1,10
Посредством аккумулятора	1,21	1,17	1,06	1,01	1,01
Посредством стека на регистрах и безадресных команд	1,29	1,23	1,10	1,04	1,03
Посредством стека в памяти и безадресных команд	1,29	1,23	1,82	1,99	1,91
Посредством стека на регистрах и одноадресных команд	1,28	1,22	1,09	1,03	1,03
Посредством стека в памяти и одноадресных команд	1,28	1,22	1,66	1,77	1,72
«Память-память» с использованием двухадресных команд	1,04	1,04	1,08	1,09	1,09
«Память-память» с использованием трехадресных команд	1,03	1,04	1,02	1,01	1,02
«Память-память» с использованием двух- и трехадресных команд	1	1	1	1	1

ресация посредством регистров и сравнительно плохие — адресация посредством стека в памяти. При сравнении по обоим параметрам S и M наилучшими оказываются типы адресации «память-память» с использованием трехадресных, а также двух- и трехадресных команд. К сделанным выводам следует относиться с осторожностью, поскольку они справедливы по отношению к параметрам в табл. 4.9 и не связаны с особенностями практической реализации адресации того или иного типа; анализ конкретной архитектуры машины и конкретного варианта ее реализации может привести к иным результатам. К подобным выводам приходят авторы и другого исследования [56], где показано, что совместное использование различных типов адресации может привести к дальнейшему совершенствованию архитектуры вычислительной машины.

КАЧЕСТВЕННЫЙ АНАЛИЗ

Относительно сравнительных характеристик команд с различными типами адресации к данным можно высказать еще несколько замечаний. Так, можно рассмотреть их влияние на

взаимосвязь оптимизации работы компилятора и архитектуры машины. Указанная оптимизация в значительной мере не зависит от архитектуры и может выполняться над программой, представленной на языке высокого уровня. Одной из сторон такой оптимизации, выполняемой с той или иной полнотой реализации ее возможностей, является типичная для большинства компиляторов *регистровая оптимизация*, т. е. минимизация числа обращений к памяти путем удержания данных (например, численных значений переменных или фрагментов арифметических выражений) в регистрах на протяжении выполнения целого ряда команд. Сама природа такой оптимизации делает ее неприменимой для машин, использующих адресацию посредством аккумулятора или стека, а также адресацию типа «память-память». Однако для машин с адресацией посредством регистров общего назначения регистровая оптимизация позволяет улучшить значение параметра M . Но прежде чем ее использовать, необходимо принять во внимание следующие обстоятельства: во-первых, стремление к более высокой модульности представления алгоритма решения задачи в машине снижает возможный эффект регистровой оптимизации; во-вторых, использование быстродействующей кэш-памяти, управляемой аппаратными средствами, позволяет добиться во многом эффекта регистровой оптимизации.

Доводом против использования адресации посредством регистров, стека или аккумулятора является наличие типичных для них ограничений на размеры операндов; последние должны иметь единственный фиксированный размер или в лучшем случае небольшое число заранее установленных размеров.

Применению адресации посредством стека и в регистрах внутри процессора препятствует сложность ее реализации на практике. Принцип адресации с помощью стека обычно предполагает отсутствие ограничений на его объем. Обычным практическим решением этой проблемы является применение небольшого набора регистров в процессоре и использование области основной памяти в качестве остальной части стека. Например, ЭВМ HP3000 [15] содержит четырехрегистровый стек в процессоре. Если в него необходимо загрузить более четырех данных, нижние данные пересылаются из регистров в основную память. Следствием этого является увеличение числа пересылок между памятью и процессором при вычислении значений сложных выражений.

Наконец, необходимо напомнить о наличии недостатков, собственных архитектуре машины, использующей регистры общего назначения. Некоторые из них, в частности функциональное несоответствие регистров машины и представления данных в языках программирования, а также трудности отлад-

ки программ, относятся в равной мере и к архитектуре машин, использующих стеки и аккумуляторы.

ОПЕРАЦИИ ВЫСОКОГО УРОВНЯ

Наиболее очевидным решением проблемы сокращения семантического разрыва между архитектурой машины и языком программирования является введение в состав набора команд машины команд высокого уровня. Последние способны выполнять операции и функции, описываемые языками высокого уровня. Некоторые основные положения, уже изложенные в данной главе, можно рассматривать как значительный шаг вперед в указанном направлении, поскольку они позволяют предоставить машине такие возможности, как выполнение операций по преобразованию данных, индексированию массивов, вызову процедур, синхронизации вычислительных процессов и т. д. Сфера применения операций высокого уровня включает обработку строк символов, арифметические операции, обработку функциональных элементов программ (например, итерационных процессов), редактирование программ, работу с ошибками, отслеживание выполнения программ.

Обработка строк символов приобретает особое значение вследствие ее широкого использования во многих программах. Например, анализ работы компилятора языка АЛГОЛ-60 вычислительной системы Burroughs B5500 [14] показал, что более трети времени выполнения программ затрачивается на операции над строками символов. Такие операции, типичные для многих языков программирования, включают манипулирование частями строк, поиск заданной части строки в строках, соединение строк и определение длины строк.

Набор арифметических операций, включаемых в архитектуру машины, должен зависеть от используемого языка (языков) программирования и «внешнего мира» машины (решаемых задач). Если класс решаемых задач ориентирован на числовые расчеты, то целесообразно введение таких операций, как извлечение корня, вычисление экспоненты и тригонометрических функций, быстрое преобразование Фурье и т. п.

Рассматриваемая в последующих главах архитектура машин B1700/1800 и SWORD является хорошей иллюстрацией использования операций высокого уровня.

МНОГОУРОВНЕВАЯ ЛЕКСИЧЕСКАЯ АДРЕСАЦИЯ

Использование стеков для управления подпрограммами оказывает влияние на способы адресации к памяти. Например, опираясь в записи активации не имеет фиксированного адреса, по-

скольку запись активации может появляться в различных областях стека в процессе выполнения программы. Вот почему в ЭВМ M68000 используется относительная адресация с помощью указателя кадра. Однако средства адресации этой машины позволяют обращаться прямо только к содержимому последнего (текущего) кадра или записи активации, в то время как возможности архитектуры машины, адекватные средствам блочно-структурированных языков программирования, должны до-

	Address	Reference
procedure UUU is		
A : INTEGER;	(0,1)	
B : INTEGER;	(0,2)	
procedure VVV is	(0,3)	
C : INTEGER;	(1,1)	
procedure XXX is	(1,2)	
D : INTEGER;	(2,1)	
begin		
C := 8;		(1,1)
A := D;		(0,1), (2,1)
XXX;		(1,2)
end XXX;		
begin		
XXX;		(1,2)
end VVV;		
procedure WWW is	(0,4)	
E : INTEGER;	(1,1)	
begin		
E := 8;		(1,1)
B := 8;		(0,2)
end WWW;		
begin		
VVV;		(0,3)
WWW;		(0,4)
end UUU;		

Рис. 4.12. Пример использования многоуровневой лексической адресации.

пускать адресацию к содержимому внешних записей активации. Одним из путей решения этой проблемы является использование *многоуровневой лексической адресации*.

Адрес лексического уровня представляет собой адресную пару, составленную из двух элементов: принадлежащего программе лексического уровня переменной или операнда и индекса (последовательного номера) этого операнда внутри данного лексического уровня. Поскольку лексическое упорядочение программы и индексы операндов остаются статическими в течение выполнения программы (хотя физическое местоположение операндов может изменяться), адрес лексического уровня можно использовать в машинных командах как адрес операнда. Информация, необходимая для динамически преобразуемых адресов лексических уровней, может формироваться и поддерживаться машиной в физических (реальных) адресах.

Этот принцип адресации лучше всего проиллюстрировать на примере. Для этой цели воспользуемся приведенной здесь про-

граммой на языке Ада (рис. 4.12). Внешняя процедура находится на лексическом уровне 0. На этом уровне определены четыре идентификатора: А, В, VVV и WWW (А и В — переменные, VVV и WWW — имена процедур; VVV и WWW могут представлять собой дескрипторы соответствующих процедур). Согласно комментариям справа от текста программы, адресами

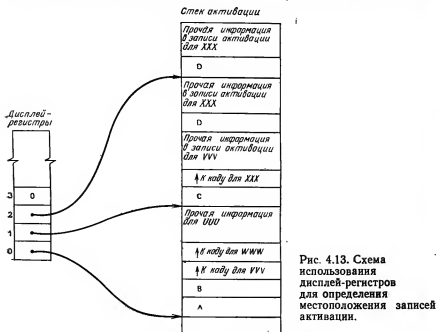


Рис. 4.13. Схема использования дисплей-регистров для определения местоположения записей активации.

лексического уровня этих идентификаторов являются (0,1), (0,2), (0,3) и (0,4).

Поскольку на лексическом уровне 1 имеются две процедуры, принадлежащие им идентификаторы имеют адресные пары вида (1, x). Подобным же образом одна из процедур имеет в свою очередь вложенный блок, содержащий адресные пары в форме (2, x).

Если сравнивать однокадровую адресацию машины 68000 с многоуровневой лексической адресацией, то можно сказать, что последняя функционирует путем поддержания массива указателей кадров или записей активации. Этот массив часто называют набором *дисплей-регистров*. Дисплей-регистр указывает местоположение записи активации, соответствующей блоку программы. Часть адреса, содержащая номер лексического уровня, используется для выбора дисплей-регистра; содержимое этого регистра добавляется к индексной части адреса для полу-

чения физического адреса. Заметим, что в случае вызова рекурсивной процедуры дисплей-регистры не обязательно указывают на местоположение смежных записей активации.

Для иллюстрации использования дисплей-регистров положим, что упомянутая выше программа выполнила следующую последовательность операторов:

VVV;/XXX;/C:=8;/A:=D;/XXX;

Содержимое стека активации показано на рис. 4.13. По мере того как записи добавляются в стек активации или удаляются из него, машина обновляет содержимое дисплей-регистров таким образом, чтобы каждый из них указывал местоположение записи активации, соответствующей каждому лексическому уровню. Машина «разрешает» (вычисляет) адрес (1, 1) для машинной команды (команд), соответствующей оператору C:=8, путем добавления единицы к содержимому дисплей-регистра 1. Для «разрешения» адреса (0,1) она добавляет единицу к содержимому дисплей-регистра 0.

Хотя многоуровневая лексическая адресация и включена в рассмотрение, ее применение в будущем остается под вопросом. Одной из причин таких сомнений является тот факт, что ее основное назначение — решение проблемы определения областей действия имен, например ситуаций, когда некоторые переменные используются во внутреннем блоке, не будучи в нем описанными, и поэтому совместно используются как глобальные с некоторым внешним блоком. А именно это в настоящее время считается многими разработчиками языков программирования и системными программистами нежелательным.

УПРАЖНЕНИЯ

4.1. Какая конструкция языка программирования может препятствовать реализации на практике архитектуры машины с теговой памятью?

4.2. Чтобы удостовериться в том, что принцип организации памяти посредством тегов сокращает, а не увеличивает потребность в памяти, вычислите значение параметра R (среднее число обращений к операнду в машинных командах), воспользовавшись имеющейся в вашем распоряжении программой. Поскольку анализу подлежат не операторы исходной программы, а генерируемые машинные команды, воспользуйтесь компилятором, который может «выдать» листинг программы на языке ассемблера.

4.3. Используя такой же подход, как и при анализе требований к памяти с теговой организацией, положим, что машина X располагает командами, имеющими 8-битовый код операции и два 4-битовых поля, задающих длину операндов (подобный формат в Системе 370 имеют команды десятичной арифметики и обработки строк символов, используемые, например, при выполнении программ на языке КОБОЛ). Положим также, что некоторая машина Y имеет команды, инвариантные к типу обрабатываемых данных и состоящие из 6-битового кода операции при условии, что каждый операнд, располагающийся в памяти, имеет 8-битовый тег (определяющий тип данных и

размер занимаемой ими области памяти). Пусть каждая команда обращается к двум операндам, причем на каждый операнд в среднем приходится R обращений. Определите, при каком значении R машины Y требуется меньший объем памяти.

4.4. Воспользуемся исходными данными упражнения 4.3 и положим, что $R=10$, адреса операндов занимают 16 бит, средний размер операнда равен 32 бит (не считая тега). Определите разницу объема памяти, занимаемого одной и той же программой в машинах X и Y .

4.5. Сформулируйте условия выполнения команды EXECUTE Системы 370, если бы память этой вычислительной системы имела теговую организацию.

4.6. При описании одноуровневой памяти упоминалось, что хотя модель этой памяти и допускает включение в нее устройств ввода-вывода без памяти, осуществление этого сопровождается появлением «двусмысленностей» в правилах функционирования такой модели. Например, для представления в такой модели устройства чтения перфокарт необходимо определение специального 80-символьного поля в качестве «вершины» очереди, образованной записями, держателем которых является устройство чтения перфокарт. Обращение к этому полю имеет эффект чтения очередной карты, поскольку содержимое поля представляет собой содержимое карты. Проанализируйте недостатки такой модели.

4.7. Для оператора $A:=(B+C)*(D+E)$ определите команды, которые генерируют машины, использующие адресацию посредством регистров, аккумулятора, стека с безадресными или одноадресными командами, а также адресацию типа «память-память» с двух-, трехадресными или совместно двух- и трехадресными командами.

ЛИТЕРАТУРА

1. Feustel E. A., On the Advantages of Tagged Architecture, *IEEE Transactions on Computers*, C-22(7), 644—656 (1973).
2. Palmer J., The Intel 8087 Numeric Data Processor, *Proceedings of the Seventh International Symposium on Computer Architecture*, New York, ACM, 1980, pp. 179—181.
3. Steele G. L., Jr., Arithmetic Shifting Considered Harmful, *SIGPLAN Notices*, 12(11), 61—69 (1977).
4. Hehner E. C. R., Computer Design to Minimize Memory Requirements, *Computer*, 9(8), 65—70 (1976).
5. Chevance R. J., Heidet T., Static Profile and Dynamic Behavior of Cobol Programs, *SIGPLAN Notices*, 13(4), 44—57 (1978).
6. Ibbett R. N., Capon P. C., The Development of the MU5 Computer System, *Communications of the ACM*, 21(1), 13—24 (1978).
7. Spier M. J., A Model Implementation for Protective Domains, *International Journal of Computer and Information Sciences*, 2(3), 201—229 (1973).
8. Spier M. J., An Experimental Implementation of the Kernel/Domain Architecture, *Operating System Review*, 7(4), 8—21 (1973).
9. Spier M. J., A Pragmatic Proposal for the Improvement of Program Modularity and Reliability, *International Journal of Computer and Information Sciences*, 4(2), 133—149 (1975).
10. Linden T. A., Operating System Structures to Support Security and Reliable Software, *Computing Surveys*, 8(4), 409—445 (1976).
11. Myers G. J., *Composite/Structured Design*, New York, Van Nostrand Reinhold, 1978.
12. Steel R., Another General Purpose Computer Architecture, *Computer Architecture News*, 5(8), 5—11 (1977).
13. Keedy J. L., A Technique for Passing Reference Parameters in an Information-Hiding Architecture, *Computer Architecture News*, 7(9), 11—15 (1979).

14. Batson A. P., Brundage R. E., Kearns J. P., Design Data for Algol-60 Machines, Proceedings of the Third Annual Symposium on Computer Architecture, New York, ACM, 1976, 151—154.
15. Blake R. P., Exploring a Stack Architecture, *Computer*, 10(5), 30—39 (1977).
16. Dennis J. B., Van Horn E. C., Programming Semantics for Multiprogrammed Computers, *Communications of the ACM*, 9(3), 143—155 (1966).
17. Denning P. J., Fault-Tolerant Operating Systems, *Computing Surveys*, 8(4), 359—390 (1976).
18. Lampson B. W., Dynamic Protection Structures, Proceedings of the 1969 Fall Joint Computer Conference, Montvale, N. J., AFIPS Press, 1969, pp. 27—38.
19. Lampson B. W., Protection, Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton, N. J., Princeton University, 1971, pp. 437—443; reprinted in *Operating System Review*, 8(1), 18—24, (1974).
20. Gligor V. D., Review and Revocation of Access Privileges Distributed through Capabilities, IEEE Transactions on Software Engineering, SE-5(6), 575—586 (1979).
21. Lampson B. W., Sturgis H. E., Reflections on an Operating System Design, *Communications of the ACM*, 19(5), 251—265 (1976).
22. Cohen E., Jefferson D., Protection in the Hydra Operating System, Proceedings of the Fifth Symposium on Operating System Principles, New York, ACM, 1975, pp. 141—160.
23. England D. M. Architectural Features of System 250, Infotech State of the Art Report 14, Operating Systems, Berkshire, England, Infotech, 1972, pp. 395—428.
24. IBM System/38 Technical Developments, Atlanta, IBM, 1978.
25. Needham R. M., Walker R. D. H., The Cambridge CAP Computer and Its Protection System, Proceedings of the Sixth Symposium on Operating System Principles, New York, ACM, 1977, pp. 1—10.
26. Herbert A. J., A New Protection Architecture for the Cambridge Capability Computer, *Operating System Review*, 12(1), 24—28 (1978).
27. Myers G. J., Storage Concepts in a Software-Reliability-Directed Computer Architecture, Proceedings of the Fifth Annual Symposium on Computer Architecture, New York, ACM, 1978, pp. 107—113.
28. Myers G. J., SWARD — A Software-Oriented Architecture, Proceedings of the International Workshop on High-Level Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 163—168.
29. Battarel G. J., Cheavance R. J., Design of a High Level Language Machine *Computer Architecture News*, 6(9), 5—17 (1978).
30. Saal H. J., Gat I., A Hardware Architecture for Controlling Information Flow, Proceedings of the Fifth Annual Symposium on Computer Architecture, New York, ACM, 1978, pp. 73—77.
31. Fabry R. S., Capability-Based Addressing, *Communications of the ACM*, 17(7), 403—412 (1974).
32. Saltzer J. H., Schroeder M. D., The Protection of Information in Computer Systems, Proceedings of the IEEE, 63(9), 1278—1308 (1975).
33. Gligor V. D., Architectural Implications of Abstract Data Type Implementation, Proceedings of the Sixth Annual Symposium on Computer Architecture, New York, ACM, 1979, pp. 20—30.
34. Saal H. J., Gat I., A Hardware Architecture for Controlling Information Flow, Proceedings of the Fifth Annual Symposium on Computer Architecture, New York, ACM, 1978, pp. 73—77.
35. Gehringer E. F., Variable-Length Capabilities as a Solution to the Small-Object Problem, Proceedings of the Seventh Symposium on Operating System Principles, New York, ACM, 1979, pp. 131—142.
36. Lam C., Madnick S. E., Properties of Storage Hierarchy Systems with Mul-

- multiple Page Sizes and Redundant Data, *ACM Transactions on Database Systems*, 4(3), 345—367 (1979).
37. Hoare C. A., Monitors, An Operating System Structuring Concept, *Communications of the ACM*, 17(10), 549—557 (1974).
 38. Hoare C. A., Communicating Sequential Processes, *Communications of the ACM*, 21(8), 666—677 (1978).
 39. Brinch Hansen P., Distributed Processes, A Concurrent Programming Concept, *Communications of the ACM*, 21(11), 934—941 (1978).
 40. Dijkstra E. W., Cooperating Sequential Processes, in F. Genuys, Ed., *Programming Languages*, London, Academic, 1968, pp. 43—112.
 41. Bloom T., Evaluating Synchronization Mechanisms, *Proceedings of the Seventh Symposium on Operating System Principles*, New York, ACM, 1979, pp. 24—32.
 42. Atkinson R. R., Hewitt C. E., Specification and Proof Techniques for Serializers, *IEEE Transactions on Software Engineering*, SE-5(1), 10—23 (1979).
 43. Reed D. P., Kanodia R. K., Synchronization with Eventcounts and Sequencers, *Communications of the ACM*, 22(2), 115—123 (1979).
 44. Reference Manual for the Ada Programming Language, Proposed Standard Document, U. S. Department of Defense, 1980.
 45. van de Snepscheut J. L., Introducing the Notion of Processes to Hardware, *Computer Architecture News*, 7(7), 13—23 (1979).
 46. Ford W. S., Hamacher V. C., Hardware Support for Inter-Process Communication and Processor Sharing, *Proceedings of the Third Annual Symposium on Computer Architecture*, New York, ACM, 1976, pp. 113—117.
 47. Atkinson T. D., Architecture of Series 60/Level 64, *Honeywell Computer Journal*, 8(2), 94—106 (1974).
 48. Atkinson T. D. et al., Modern Central Processor Architecture, *Proceedings of the IEEE*, 63(6), 863—870 (1975).
 49. Myers G. J., The Case Against Stack-Oriented Instruction Sets, *Computer Architecture News*, 6(3), 7—10 (1977).
 50. Keedy J. L., On the Use of Stacks in the Evaluation of Expressions, *Computer Architecture News*, 6(6), 22—28 (1978).
 51. Myers G. J., The Evaluation of Expressions in a Storage-to-Storage Architecture, *Computer Architecture News*, 6(9), 20—23 (1978).
 52. Keedy J. L., On the Evaluation of Expressions Using Accumulators, Stacks, and Storage-to-Storage Instructions, *Computer Architecture News*, 7(4), 24—25 (1978).
 53. Keedy J. L., More on the Use of Stacks in the Evaluation of Expressions, *Computer Architecture News*, 7(8), 18—22 (1979).
 54. Tanenbaum A. S., Implications of Structured Programming for Machine Architecture, *Communications of the ACM*, 21(3), 237—246 (1978).
 55. Elshoff J. L., An Analysis of Some Commercial PL/I Programs, *IEEE Transactions on Software Engineering*, SE-2(2), 113—120 (1976).
 56. Belgard R. A., Schneider V. B., A Comparison of the Code Space and Execution Time Required for Fortran Assignment Statements on Six Computer Architectures, *Micro11 Proceedings*, New York, ACM, 1978, pp. 56—64.
 57. Rattner J., Lattin W. W., Ada Determines Architecture of 32-bit Microprocessor, *Electronics*, 4(54), 119—126 (1981).
 58. Myers G. J., Buckingham B. R. S., A Hardware Implementation of Capability-Based Addressing, *Operating Systems Review*, 14(4), 13—25 (1980).
 59. Eventoff W., Harvey D., Price R. J., The Rendezvous and Monitor Concepts, Is There an Efficiency Difference? *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, New York, ACM, 1980, pp. 156—165.
 60. Berstis V., Security and Protection of Data in the IBM System/38, *Proceedings of the Seventh Annual Symposium on Computer Architecture*, New York, ACM, 1980, pp. 245—252.

ЧАСТЬ II

АРХИТЕКТУРА, ОРИЕНТИРОВАННАЯ НА ЯЗЫК ПРОГРАММИРОВАНИЯ

ГЛАВА 5

УЧЕБНАЯ ПЛ-МАШИНА

Чтобы проиллюстрировать основные идеи, изложенные в части I книги, рассмотрим учебную ПЛ-машину¹⁾ [1]. Первоначально эта машина предназначалась для демонстрации возможности оптимизации, или настройки, архитектуры ЭВМ, ориентированной на язык программирования. Это обстоятельство является причиной некоторых особенностей данной машины. Во-первых, машина физически не реализована и существует только на бумаге. Во-вторых, ее архитектура не является завершенной, цельной: отсутствуют, например, некоторые функции, которые могли бы понадобиться при нормальной работе операционной системы. В данной главе рассматривается первоначальная неоптимизированная версия архитектуры, а оптимизация архитектуры такого типа, как пример оптимизации архитектуры ЭВМ, описывается в гл. 20. Перечисленные выше особенности не препятствуют достижению сформулированных выше целей, поскольку учебная ПЛ-машина оказывается простой и удобной для иллюстрации многих основных положений, рассматриваемых в части I книги.

Архитектура ПЛ-машины ориентирована на язык программирования, являющийся подмножеством языка ПЛ/1, которое не полностью совместимо с базовым языком. Пользуясь терминологией, введенной в гл. 4, можно сказать, что эта машина имеет память с теговой организацией, дескрипторы информационных объектов, стеки для вычисления выражений, средства работы с подпрограммами, управляемую языком ПЛ/1 память, многоуровневую лексическую адресацию.

УЧЕБНЫЙ ЯЗЫК ПЛ

Поскольку архитектура учебной ПЛ-машины ориентирована на учебный язык ПЛ, знакомство с машиной лучше всего начать с краткого описания структуры этого языка. В табл. 5.1 приве-

¹⁾ Сочетание ПЛ происходит от английского PL (Programming Language), что означает «язык программирования». — *Прим. перев.*

Таблица 5.1. Учебный язык ПЛ

Операторы	Типы данных	Встроенные функции	Операции
PROCEDURE BEGIN RETURN END DO DO WHILE DO CASE Оператор итератив- ного цикла DO CALL GO TO DECLARE Оператор присваива- ния IF ALLOCATE FREE	FIXED FLOAT BIT CHARACTER LABEL ENTRY Scalar Array	ABS ATAN BIT CHARACTER COS DUMP E-FORMAT END-OF-DATA EXP FIXED FLOAT INDEX LENGTH LOG MAX MIN MOD RANDOM SIGN SIN SQRT SUBSTR TAN TIME UNDEFINED	+ - • / ! < > = <= >= <> <- >=

дены операторы, типы данных, встроенные функции и знаки операций учебного языка ПЛ.

Тип данных FIXED эквивалентен типу FIXED BINARY (31,0) языка ПЛ/1, тип данных FLOAT эквивалентен типу FLOAT BINARY (21), BIT аналогичен BIT (1), а CHARACTER — CHARACTER (4095) VARYING.

Все переменные, за исключением массивов, размещаются в памяти автоматически. Все массивы размещаются в памяти, управляемой языком ПЛ/1, т. е. в области (или областях памяти), выделяемой и освобождаемой с помощью операторов ALLOCATE и FREE.

Операторы учебного языка ПЛ достаточно точно соответствуют аналогичным операторам языка ПЛ/1. Программа может быть разделена на части с помощью блоков типа BEGIN, а также внутренних процедур и процедур-функций. Оператор GO TO рассматриваемого языка имеет более ограниченные возможности, чем аналогичный оператор языка ПЛ/1; адресат, которому он передает управление, не может находиться внутри со-

ставных операторов (например, предложения THEN или ELSE, цикла DO).

При определении значений арифметических и логических выражений, а также строк символов здесь, как и в языке ПЛ/1, используются весьма гибкие правила автоматического преобразования данных. Правила выполнения арифметических операций над массивами в учебном языке ПЛ такие же, как и в языке ПЛ/1.

Учебный язык ПЛ представляет пользователю также 25 встроенных функций, подобных аналогичным функциям языка ПЛ/1. Как и в языке ПЛ/1, некоторые из этих функций могут использоваться в качестве псевдопеременных, например появляться в первой части оператора присваивания. Встроенная функция INPUT и псевдопеременная OUTPUT семантически эквивалентны операторам GET LIST и PUT LIST языка ПЛ/1. Функции BIT, CHARACTER, FIXED и FLOAT используются для преобразования данных, функции INDEX, LENGTH и SUBSTR — для работы со строками данных, функция UNDEFINED — для проверки того, задано ли значение переменной.

СТРУКТУРА ПАМЯТИ УЧЕБНОЙ ПЛ-МАШИНЫ

Изучение архитектуры машины лучше всего начинать с принципов организации ее памяти. С точки зрения разработчика компилятора и программиста, составляющего программы на машинном языке, учебная ПЛ-машина содержит, как показано на рис. 5.1, семь различных запоминающих устройств (каждое из которых называется также памятью), четыре специальных регистра и набор из 16 регистров. Каждое запоминающее устройство или регистр имеет определенное назначение. Все они между собой логически связаны, благодаря тому, что информация, находящаяся в одних из них, указывает местоположение соответствующей информации в других. (Существующие связи показаны на рис. 5.1 стрелками).

В учебной ПЛ-машине не реализован принцип «основной памяти». Наибольший интерес представляет *стековая память данных* (data stack memory). Нижняя часть этой памяти используется как стек (принцип работы которого «поступивший первым обрабатывается последним») для вычисления выражений и размещения информации о местоположении выполняемых в текущее время процедур, функций и блоков BEGIN. *Регистр — указатель стековой памяти данных* всегда содержит информацию об адресе слова, находящегося на вершине этого стека. Верхняя часть стековой памяти данных управляется ап-

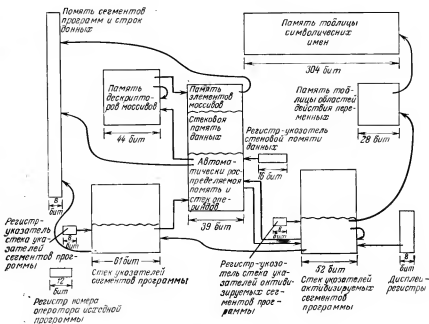


Рис. 5.1. Структура памяти учебной ПЛ-машин.

паратными средствами; она используется для размещения элементов массивов.

Стековая память данных — это так называемая теговая память. Слово этой памяти длиной 39 бит состоит из 7-битового поля, содержащего тег (указатель типа данных), и 32-битового поля, включающего данные (рис. 5.2). Если у тега бит и равен 1, то слово содержит неопределенную величину; если же единичное значение имеет бит p или a , то слово — указатель процедуры (функции) или массива соответственно. Остальные четыре бита тега содержат информацию о типе данных.

Содержимое 32-битового поля данных интерпретируется в зависимости от значения тега (табл. 5.2). В некоторых случаях это поле интерпретируется как состоящее из двух отдельных полей длиной 12 и 20 бит. Содержимое этих полей принято указывать в скобках.

С целью упрощения объяснения назначения некоторых из рассматриваемых слов памяти массивы представлены как указатели элементов массивов, строки символов как дескрипторы строк символов, адреса переменных как дескрипторы косвенной адресации, адреса элементов массивов как дескрипторы элементов массивов и адреса сегментов объектной программы как

Слово стековой памяти данных:

1	1	1	4	12	20
u	r	a	Тип данных:		

Элемент памяти дескрипторов массива
для двумерного массива

16	16	12
Указатель первого эле- мента массива в стековой памяти данных:	Размерность	
Нижняя граница индекса	Верхняя граница индекса	Масштабный множитель
Нижняя граница индекса	Верхняя граница индекса	Масштабный множитель

Указатель дескрип-
тора массива для
предыдущего рас-
пределения памяти
для массива

Слова стена указателей сегментов программы

12	20	13	16
Длина сегмента	Указатель i-й команды сег- мента в памяти сегментов программ и строк данных:	Смещение от начала сегмента до следую- щей команды, подго- товленной к выполнению	Содержимое регистра-указа- теля стековой памяти дан- ных во время выполнения i-й команды сегмента

Слово памяти таблицы символических имен

256	8	8	32
Символическое имя	Размерность	Тип	Начальное значение

Слово памяти таблицы областей действия переменных

8	8	8	4
Указатель пер- вого элемента таблицы симбо- лических имен	Количество эле- ментов табли- цы символиче- ских имен	Количество параметров	Номер лексичес- кого уровня

Слово стена указателей активизируемых сегментов программы

8	8	16	8	12
Указатель элемента в таблице обла- стей действия переменных	Указатель эле- мента данного стена для пре- дыдущего лек- сического уровня	Значение указателя сте- ковой памяти данных во время перехода на данный лексический уровень	Значение ука- зателя стена указателей сег- ментов прог- раммы во вре- мя перехода...	Содержимое регистра номера оператора ис- ходной программы во время перехода на дан- ный лексический уровень

Рис. 5.2. Форматы слов памяти учебной ПЛ-машины.

Таблица 5.2. Содержимое слов стековой памяти данных

Метка	Содержимое поля данных
бит $u = 1$	Неопределенная величина
бит $p = 1$	Дескриптор сегмента программы (длина сегмента, адрес сегмента в памяти сегментов программ и строк данных)
бит $a = 1$	Указатель массива (не используется, адрес элемента в памяти дескрипторов массивов)
тип = fixed	Двоичное целое со знаком
тип = float	Двоичное число со знаком, представленное в форме с плавающей точкой
тип = bit	Булева величина («истинно» или «ложно»)
тип = char	Дескриптор строки символов (длина строки, адрес строки в памяти сегментов программ и строк данных)
тип = labcl ¹⁾	Дескриптор метки, представляющей собой константу (смещение в сегменте программы, адрес сегмента программы в памяти сегментов программ и строк данных)
тип = labv ²⁾	Дескриптор метки, представляющей собой переменную (смещение в сегменте программы, адрес сегмента программы в памяти сегментов программ и строк данных)
тип = star	Значение отсутствует (индекс массива задан в виде *)
тип = ind	Дескриптор косвенного адреса (адрес элемента стека указателей областей памяти активных сегментов программы, смещение относительно содержимого регистра — указателя стековой памяти данных)
тип = arlm	Дескриптор элемента массива (адрес элемента в памяти дескрипторов массивов, смещение элемента в массиве)

¹⁾ Label constant. — Прим. перев.²⁾ Label varying. — Прим. перев.

дескрипторы сегментов программы. Слово star («звездочка») используется для указания на представление индекса массива знаком *, как это делается в выражениях языка ПЛ/1, а именно $A(*, I)$.

Память сегментов программ и строк данных (string segment memory) состоит из слов длиной 8 бит и содержит объектные программы и строки символов. Объектные программы хранятся в виде одного или нескольких сегментов. Отдельный сегмент может представлять процедуру, функцию, блок BEGIN, тело цикла DO, предложение THEN или ELSE оператора IF.

Память дескрипторов массивов (array descriptor memory) содержит дескрипторы элементов массивов. Очередной дескриптор «вводится» в эту память аппаратно каждый раз при динамическом размещении массива. Каждый дескриптор представлен $D+1$ смежными словами памяти, где D — размерность массива. На рис. 5.2 показан дескриптор двумерного массива. Первое поле содержит адрес первого элемента массива в стековой памяти данных. Последующие слова содержат информацию о

каждом измерении массива, т. е. определяют минимальное и максимальное граничные значения индекса и значение масштабного множителя, используемого для вычисления расстояния между элементами массива в этом измерении. Третье поле в первом слове дескриптора используется для работы с памятью, управляемой языком ПЛ/1. (Например, если массив А уже размещен, а затем размещается опять, то существуют два дескриптора массива А, причем в текущем дескрипторе в данном поле формируется ссылка на предыдущий дескриптор.)

Стек указателей сегментов программы (program segment stack) используется для размещения информации об активных (выполняющихся в данный момент) сегментах программы. Исходная программа разделяется компилятором на один или несколько сегментов. Каждая процедура, функция, блок BEGIN, предложение THEN, предложение ELSE, тело цикла DO представляются в виде отдельных сегментов программы. Причины представлений предложений THEN, ELSE и цикла DO как отдельных сегментов объясняются в следующей главе; там же показано, что это является недостатком архитектуры данного типа.

Указатель сегмента программы помещается в стек всякий раз, когда сегмент начинает выполняться; указатель, находящийся на вершине стека, удаляется, как только завершается выполнение сегмента. Регистр — указатель стека указателей сегментов программы всегда адресует к элементу, находящемуся на вершине этого стека. Пользуясь терминологией, принятой для Системы 370, все содержимое стека указателей сегментов программы, в том числе и элемент, находящийся на вершине стека, можно рассматривать как PSW¹⁾ машины, поскольку оно определяет текущее состояние программы. Четыре поля стека указателей сегментов программы, показанные на рис. 5.2, достаточно понятны и не требуют дополнительных объяснений. Последнее поле предназначено для размещения величины, которая во время выполнения первой команды данного сегмента находится в регистре — указателе стековой памяти данных.

Исходное содержимое *памяти таблицы символических имен* (symbol table memory) устанавливается компилятором или загрузчиком программ. Эта память используется машинной при выделении локальной памяти для сегмента.

Каждому элементу таблицы символических имен соответствует идентификатор или константа исходной программы. Формат элемента таблицы представлен на рис. 5.2. Восьмибитовое поле типа используется для присвоения значения тегу слова стека данных при размещении в этой памяти очередной переменной.

¹⁾ PSW (Program status word) — слово состояния программы. — Прим. перев.

Исходное содержимое *памяти таблицы областей действия переменных* (scope table memory) также устанавливается компилятором или загрузчиком программ. Таблица содержит по одному элементу для каждого перехода на новый лексический уровень программы, т. е. по одному элементу для каждой процедуры, функции и блока BEGIN. Как показано на рис. 5.2, элемент таблицы областей действия переменных указывает элементы таблицы символических имен, связанные с данным лексическим уровнем, определяет число параметров, которые должны быть приняты, и сам уровень.

Последним типом памяти учебной ПЛ-машинны является *стек указателей активизируемых сегментов программы* (scope stack), используемый для хранения информации о процедурах, функциях и блоках BEGIN, к которым имело место обращение. Очередной элемент помещается в этот стек всякий раз, когда выполняется подобное обращение.

Как показано на рис. 5.2, элемент, помещенный в этот стек, указывает на соответствующий элемент таблицы областей действия переменных и элемент данного стека для предыдущего лексического уровня программы, а также содержит величины, находящиеся в регистрах — указателях стековой памяти данных, стека указателей сегментов программы и в регистре номера оператора исходной программы во время перехода на данный лексический уровень программы.

Машина также содержит набор из 16 *дисплей-регистров*, используемых, как показано в гл. 4, для разрешения адресных ссылок на лексическом уровне. Дисплей-регистры пронумерованы от 0 до 15; дисплей-регистр p содержит адрес элемента стека указателей активизируемых сегментов программы, соответствующих активному лексическому уровню программы p .

Последний регистр — *регистр номера оператора исходной программы* — играет пассивную роль. Он используется для отслеживания информации о выполняемом в данный момент операторе исходной программы; таким образом, если будет обнаружена ошибка, информация об этом может быть выдана программисту на языке его исходной программы. Когда компилятор генерирует код, соответствующий, например, оператору исходной программы с номером 17, то в первую очередь формируется этот номер, который загружается в данный регистр.

УПРАЖНЕНИЯ

5.1. Как используется 4-битовое поле типа слова стековой памяти данных, если бит a имеет единичное значение (т. е. слово содержит указатель массива)?

5.2. В какую память помещает свою выходную информацию компилятор и загрузчик программ?

5.3. Содержимое какой памяти не изменяется во время выполнения программы?

5.4. Как будет представлен в памяти массив строк символьных данных?

5.5. Каково максимальное число лексических уровней в программе учебной ПЛ-машины?

5.6. Исходя из описания памяти различного назначения и регистров, определите, содержимое каких устройств изменяется при выполнении оператора CALL учебного языка ПЛ.

5.7. Почему элемент стека указателей активизируемых сегментов программы содержит явную ссылку на элемент ближайшего, более низкого лексического уровня? Не должен ли этот элемент всегда быть предыдущим словом, помещенным в стек?

ЛИТЕРАТУРА

1. Wortman D. B., A Study of Language Directed Computer Design, Ph. D. dissertation, Stanford University, Stanford CA, 1972. Also available as Report CSRG-20, University of Toronto, Toronto, 1972.

ГЛАВА 6

ТРАНСЛЯЦИЯ И ВЫПОЛНЕНИЕ ПРОГРАММЫ УЧЕБНОЙ ПЛ-МАШИНЫ

ПРИМЕР ПРОГРАММЫ

Вместо того чтобы изучать архитектуру учебной ПЛ-машины путем последовательного рассмотрения ее команд, представляется более целесообразным проанализировать примеры компилирования и выполнения программ, написанных на одном и том же или различных языках высокого уровня. Вот почему в данной главе описываются процессы компилирования и трансляции двух программ на учебном языке ПЛ. Гл. 7 содержит представленные в традиционной форме спецификации команд, к которым читатель при желании может обращаться во время изучения материала данной главы.

Учебная ПЛ-машина имеет ориентированный на работу со стеком набор команд. Первые 8 бит размещаются в поле кода операции. Большинство команд содержит только код операции, и, следовательно, их длина равна 8 бит. Некоторые команды описываются двумя 8-битовыми полями; назначение второго поля зависит от типа команды. Одна из команд (LNAME) имеет длину 24 бит.

Первый из рассматриваемых примеров — трансляция и выполнение простой процедуры (рис. 6.1). Предположим, что эта процедура — часть большой программы и находится на лексическом уровне 2. В процедуре используются две переменные A и B; адресами этих переменных данного лексического уровня являются (2, 1) и (2, 2). Процедура также содержит константы 1 и 3; этим константам назначены адреса (2, 3) и (2, 4).

В результате компилирования данной процедуры создаются сегмент объектной программы в памяти сегментов программ и строк данных, четыре элемента в таблице символических имен и один элемент в таблице областей действия переменных. Эти элементы показаны на рис. 6.1; объектная программа определяется далее (будем полагать, что она записывается в памяти, начиная с адреса 446).

Хотя теперь и следует приступить к анализу объектного кода, создаваемого в результате компилирования этой процеду-

ры, полезно рассмотреть ее выполнение. Предположим, что данная процедура вызывается другой процедурой. Это означает, что последней командой, выполнявшейся в вызывающей процедуре, является команда ENTER. Команда ENTER указывает, где в стековой памяти должна находиться локальная область памяти для процедуры X, помещает соответствующие элементы в стек указателей сегментов программы и стек указателей активизируемых сегментов программы, обновляет содержимое

110 X: PROCEDURE;

120 DECLARE A FIXED;

130 DECLARE B FLOAT;

140 B = 1;

150 A = B + 3 * B;

160 END;

Элементы таблицы символических имен

10	A	0	FIXED	
11	B	0	FLOAT	
12		0	FIXED	1
13		0	FIXED	3

Элемент таблицы областей действия переменных

8	10	4	0	2
---	----	---	---	---

Рис. 6.1. Первый пример программы учебной ПЛ-машины.

дисплей-регистров. В результате состояние машины будет иметь вид, показанный на рис. 6.2.

Теперь необходимо рассмотреть объектный код процедуры X. Для простоты представления объектный код записывается на гипотетическом языке ассемблера.

Первой командой в любой процедуре должна быть команда SCOPEID; ее операндом — адрес элемента таблицы областей действия переменных, относящегося к данной процедуре. Во время выполнения команды SCOPEID не производится абсолютно никаких действий (т. е. эта команда аналогична команде НЕТ ОПЕРАЦИИ¹⁾). Однако во время выполнения предшествующей ей команды ENTER машина обращается к команде SCOPEID для определения местоположения элемента стека указателей активизируемых сегментов; этот элемент в свою очередь определяет лексический уровень процедуры и соответствующие входы таблицы символических имен. Следовательно, первой генерируется команда

SCOPEID 8

Следующим шагом является генерирование команд для оператора 140. Первой из этих команд должна быть команда

LINE 140

¹⁾ Например, команде NOP языка ассемблера Системы 370. — Прим. перев.

по которой производится загрузка номера оператора в регистр номера оператора исходной программы. Для описания генерирования остальных команд рассмотрим обратную польскую запись оператора: $B1=$. Такая запись показывает последователь-

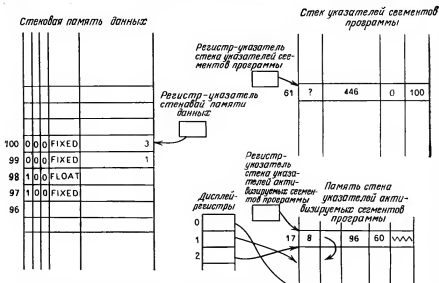


Рис. 6.2. Состояние машины при входе в процедуру X.

ность команд, которые должны быть сформированы: загрузка адреса переменной B в стек, загрузка величины 1 в стек и запись значения переменной v в память. Первой должна быть команда

SNAME 2,2

Операндом этой команды является адрес данного лексического уровня (адрес переменной B). Команда формирует дескриптор косвенного адреса и загружает его в стек. Дескриптор косвенного адреса является не полностью определенным адресом лексического уровня. Представление дескриптора длиной 32 бита включает 12-битовый адрес элемента стека указателей активизируемых сегментов программы и 20-битовое смещение относительно исходного содержимого регистра — указателя стековой памяти данных. Формирование дескриптора косвенного адреса производится очень просто. Выполняя команду SNAME x, y , машина создает дескриптор косвенного адреса, значение первых 12 бит которого равно величине x , находящей-

ся в дисплей-регистре, а остальные 20 бит — величине у. Следовательно, команда формирует дескриптор косвенного адреса, содержащий величину 17,2, в слове стековой памяти данных с адресом 101.

Следующий шаг — загрузка в стек величины 1. Это действие выполняют команды

SNAME 2,3

EVAL

Первая из этих команд формирует дескриптор косвенного адреса, содержащий величину 17,3, в слове стековой памяти данных с адресом 102 (регистр — указатель стековой памяти данных теперь указывает на слово с адресом 102). По команде EVAL на место дескриптора косвенного адреса, находящегося на вершине стека, помещается содержимое слова, адресуемого этим дескриптором. Чтобы выполнить такую операцию, машина обращается к слову стека указателей активизируемых сегментов с адресом 17, извлекает записанное в этом слове значение указателя стековой памяти данных (96), добавляет к нему 3 (получая 99) и пересылает содержимое слова стековой памяти данных с этим адресом (константу 1) на вершину стека. Следовательно, слово стековой памяти данных с адресом 102 содержит теперь копию слова с адресом 99.

Затем генерируется команда

STORE

Эта команда имеет дело с двумя словами, находящимися на вершине стека. Предполагается, что второе слово представляет собой дескриптор — в данном случае дескриптор косвенного адреса переменной В. Команда копирует содержимое слова, находящегося на вершине стека, в слово, указываемое дескриптором, а затем удаляет второе слово. Следовательно, слово с адресом 98 (переменная В) теперь содержит величину 1, а бит ее метки, указывающий на неопределенное содержимое слова, равен нулю, т. е. сброшен. Заметим, что данные типа FIXED записываются в слово памяти, для которого установлен тип данных FLOAT; машина распознает эту ситуацию и выполняет требуемое преобразование данных в форму с плавающей точкой.

Теперь оператор В=1 выполнен, но указатель стековой памяти данных имеет значение 101 (величина 1 все еще находится в стеке). Производится очистка стека с помощью команды

POP

которая удаляет слово, находящееся на вершине стека (т. е. вычитает 1 из указателя стековой памяти данных).

Объектный код оператора исходной программы с номером

150 генерируется аналогичным образом. Обратная польская запись оператора 150 имеет вид: $AB3B^*+=$; а генерируемый код

LINE	150	...
SNAME	2,1	дескриптор косвенного адреса переменной A, ...
SNAME	2,2	дескриптор косвенного адреса переменной B, дескриптор косвенного адреса переменной A, ...
EVAL		1, дескриптор косвенного адреса переменной A, ...
SNAME	2,4	дескриптор косвенного адреса константы 3,1, дескриптор косвенного адреса переменной A, ...
EVAL		3,1, дескриптор косвенного адреса переменной A, ...
SNAME	2,2	дескриптор косвенного адреса переменной B, 3,1, дескриптор косвенного адреса переменной A, ...
EVAL		1,3,1, дескриптор косвенного адреса переменной A, ...
MUL		3,1, дескриптор косвенного адреса переменной A, ...
ADD		4, дескриптор косвенного адреса переменной A, ...
STORE		4, ...
POP		...

Справа от записи каждой команды показано получаемое в результате ее выполнения содержимое слов стековой памяти данных, начиная с адреса 101 и выше. Наряду с выполнением своих основных функций команда EVAL проверяет также, является ли операнд операции определенной величиной. Например, если бы переменная B представляла собой неопределенную величину, то при выполнении первой команды EVAL произошло бы программное прерывание. Арифметические команды MUL и ADD выполняют свои операции над содержимым двух слов, находящихся на вершине стека, затем удаляют эти слова и загружают результат в стек. Если операнды этих команд представляют собой числовые данные различного типа (как это имеет место при выполнении команды MUL), то во время выполнения команды производится автоматическое преобразование формы представления данных. Оставшаяся часть генерируемого объектного кода имеет вид

LINE 160
END 8

По команде END производится возврат управления в вызывающую процедуру; при этом уничтожаются локальная память данных в стековой памяти данных, элементы, находящиеся на вершине стека указателей сегментов программы и стека указателей активизируемых сегментов, и обновляется содержимое дисплей-регистров. В данном случае при выполнении команды END в регистре — указателе стековой памяти данных установится значение, равное 96, в регистре — указателе стека указателей сегментов программы — значение 60, в регистре — указателе стека указателей активизируемых сегментов — значение 16 и в дисплей-регистре 2 — значение 0.

СЕГМЕНТЫ ПРОГРАММЫ ДЛЯ ОПЕРАТОРОВ IF И ЦИКЛОВ DO

Одной из необычных характеристик учебной ПЛ-машины является отсутствие команды перехода. Имеющаяся в этой машине команда GO TO соответствует оператору GO TO учебного языка ПЛ. С помощью команды GO TO управление передается по адресу, задаваемому дескриптором метки-константы или метки-переменной.

Отсутствие команды передачи управления ставит вопрос о том, как должен выглядеть объектный код программы для операторов IF и циклов DO. Разработчики архитектуры машины полагали, что оператор IF THEN ELSE можно рассматривать как условный вызов подпрограммы. Если условие, задаваемое оператором IF, выполняется, то вызывается код, соответствующий предложению THEN, если не выполняется, то действует код, соответствующий предложению ELSE. В обоих случаях управление возвращается оператору, следующему за оператором IF. Подобным же образом цикл DO можно рассматривать как вызов тела цикла, после чего производится возврат к выполнению оператора, следующего за циклом DO. В свою очередь тело цикла считается состоящим из повторяющихся проверок условия выхода из цикла DO, за которыми следует вызов кода, указанного в цикле.

Производимые действия лучше всего понять на каком-нибудь примере. Предположим, что к процедуре X, показанной на рис. 6.1, добавлен оператор

```
155  IF(A=1)  THEN B=3;
        ELSE A=3;
```

Вместо генерирования одного сегмента программы компилятор теперь сформирует три сегмента. Двумя новыми сегментами являются следующие:

SNAME 2,2
 SNAME 2,4
 EVAL
 STORE
 POP

SNAME 2,1
 SNAME 2,4
 EVAL
 STORE
 POP

Первым шагом при выполнении оператора 155 является проверка условия, указанного в операторе IF; генерируемый код начинается со следующих команд:

LINE 155
 SNAME 2,1
 EVAL
 SNAME 2,3
 EVAL
 EQ

Команда EQ сравнивает содержимое двух слов, находящихся на вершине стека, с целью выяснения, равны ли они (при этом, если необходимо, то выполняются требуемые преобразования формы представления данных), и помещает на их место на вершину стека логическую переменную. Теперь, если предварительно был подготовлен одномерный массив из двух элементов, представляющих дескрипторы двух сегментов программы, то может быть сгенерирована команда SUBS (subscript), в которой используется значение логической переменной для выбора одного из дескрипторов сегментов программы. После этого с помощью команды CALL вызывается надлежащий сегмент программы. Имя CALL (ВЫЗОВ) для данной команды выбрано неудачно, поскольку по команде CALL элемент только помещается в стек сегментов программы и производится передача управления в этот сегмент. При выполнении команды не происходит передачи параметров или изменения лексического уровня. Командой машинного уровня, соответствующей оператору CALL учебного языка ПЛ, является команда ENTER.

Для оператора 155 описанные выше команды генерируются после команды EQ:

SNAME	2,5	Формирование дескриптора косвенного адреса сегмента программы
SWAP		Обмен содержимым между двумя словами, находящимися в верхней части стека
SUBS	1	Помещение дескриптора элемента массива на вершину стека (регистр — указатель стековой памяти данных содержит теперь величину 101)

ляется, то должны быть добавлены команды **ALLOCATE** и **FREE** для выделения и освобождения памяти массива. Так как массив должен быть инициализирован, необходимо сформировать еще два элемента (дескрипторы сегментов программы) в таблице символических имен и должны быть сгенерированы команды инициализации массива. Полный листинг работы компилятора для процедуры **X**, в которую добавлен оператор **IF**, показан на рис. 6.3.

Чтобы понять, во что обходится подобная реализация операторов, предположим, что к набору команд учебной ПЛ-машины добавляют две новые команды: **BRANCH** (ПЕРЕХОД) и **BRANCH FALSE** (ПЕРЕХОД, ЕСЛИ ЛОЖНО). Обе команды имеют длину 16 бит; следующие за кодом операции 8 бит представляют собой величину со знаком, добавляемую к содержимому счетчика команд. Для команды **BRANCH FALSE** эта величина добавляется, если только логическая величина, находящаяся на вершине стека, имеет значение **FALSE**. По команде **BRANCH FALSE** из стека удаляется слово, находящееся на его вершине. С учетом этих изменений объектная программа, приведенная на рис. 6.3, может быть записана в следующем виде:

SCOPEID	8	STORE		SNAME	2,3
LINE	140	POP		EVAL	
SNAME	2,2	LINE	150	EQ	
SNAME	2,3	SNAME	2,1	BRANCHF	11
SNAME	2,2	SNAME	2,2	POP	
EVAL		EVAL		BRANCH	9
MUL		SNAME	2,4	SNAME	2,1
ADD		EVAL		SNAME	2,4
SNAME	2,2	STORE		EVAL	
SNAME	2,4	POP		STORE	
EVAL		LINE	155	POP	
STORE		SNAME	2,1	LINE	160
EVAL		EVAL		END	8

Описанные выше незначительные изменения в наборе команд уменьшают количество генерируемых команд программы с 67 до 39 и улучшают временные характеристики системы, поскольку из программы исключаются команды **SUBS**, **ALLOC**, **FREE** и **CALL**, выполнение которых занимает значительное время.

Однако отсутствие традиционных команд перехода делает реализацию циклов **DO-WHILE** и итеративных циклов **DO** еще более сложной. Для выполнения циклов **DO** требуются два уровня вызова сегментов; для этой цели предусмотрено несколько команд: **CYCLE**, **DOTEST**, **DOINCR** (см. гл. 7).

Приведенные выше критические замечания относительно реализации операторов IF и DO в учебной ПЛ-машине не следует воспринимать как критику всей машины, поскольку последняя используется только как средство для изучения некоторых новых подходов к построению архитектуры ЭВМ. Однако можно сделать вывод, что целью выбора варианта построения архитектуры является достижение точного и эффективного выполнения машинной программ, написанных на языках высокого уровня. Кроме того, следует признать неоправданным стремление разработчиков архитектуры машины исключить из набора команд команду перехода или свести к минимуму использование оператора GO TO.

ПРИМЕР ВЫЗОВА ПОДПРОГРАММЫ

Поскольку существенная часть архитектуры связана с автоматическим управлением подпрограммами и операциями над массивами данных, целесообразно проанализировать пример, в котором раскрываются соответствующие возможности архитектуры. Рассмотрим следующую программу, написанную на учебном языке ПЛ:

```
1  XYZ: PROCEDURE;  
2  DECLARE A(*) FLOAT;  
3  DECLARE (B,C) FIXED;  
4  B=4;  
5  ALLOCATE A(1:B);  
6  A=1;  
7  C=B+B*B;  
8  CALL ZZZ(B);  
9  A(B)=C;  
10 ZZZ: PROCEDURE(M);  
11   DECLARE M FIXED;  
12   C=M+M;  
13   END;  
14 END;
```

На рис. 6.4 показаны два сегмента программы, сгенерированные компилятором. Для удобства ссылок машинные команды пронумерованы. Предположим, что первый сегмент располагается в памяти сегментов программ и строк данных, начиная с адреса 200, а второй — с адреса 800. На рис. 6.5 представлена остальная информация, формируемая в результате работы компилятора, а рис. 6.6 демонстрирует состояние машины при входе в сегмент XYZ.

Команды 1—8 понятны без пояснений и подобны командам, содержащимся в предыдущем примере. Команды 9—14 предназначены для распределения памяти для массива A. По команде ALLOC на вершину стека помещается дескриптор косвенного

адреса указателя массива, за которым следуют минимальные и максимальные граничные значения индексов. Заметим, что оператор 6 исходной программы выполняет операцию над массивом: все элементы массива А устанавливаются равными 1. Эта операция реализуется с помощью команд 16—20. При выполнении команды STORE выясняется, что в массив записы-

Сегмент XYZ			Сегмент ZZZ		
1	SCOPEID 1	27	SWANE 0,2	50	SCOPEID 2
2	LINE 4	28	EVAL	51	LINE 12
3	SWANE 0,2	29	MUL	52	SWANE 0,3
4	SWANE 0,4	30	ADD	53	SWANE 1,1
5	EVAL	31	STORE	54	EVAL
6	STORE	32	POP	55	SWANE 1,1
7	POP	33	LINE 8	56	EVAL
8	LINE 5	34	SWANE 0,2	57	ADD
9	SWANE 0,2	35	SWANE 0,6	58	STORE
10	EVAL	36	ENTER 1	59	POP
11	SWANE 0,5	37	POP	60	LINE 13
12	EVAL	38	POP	61	UNDEF
13	SWANE 0,1	39	LINE 9	62	END 2
14	ALLOC 1	40	SWANE 0,1		
15	LINE 6	41	SWANE 0,2		
16	SWANE 0,1	42	EVAL		
17	SWANE 0,5	43	SUBS 1		
18	EVAL	44	SWANE 0,3		
19	STORE	45	EVAL		
20	POP	46	STORE		
21	LINE 7	47	POP		
22	SWANE 0,3	48	LINE 14		
23	SWANE 0,2	49	HALT		
24	EVAL				
25	SWANE 0,2				
26	EVAL				

Рис. 6.4. Объектный код для процедур XYZ и ZZZ.

вается скалярная величина. Следовательно, эта скалярная величина помещается в каждый элемент массива.

Команды 34—38 генерируются в результате компилирования оператора CALL. Операнд команды ENTER определяет количество аргументов вызова процедуры (фактических параметров). Дескрипторы фактических параметров и дескриптор сегмента программы должны находиться в стеке. При реализации команды ENTER проверяется соответствие количества фактических параметров процедуры количеству ее формальных параметров, изымается из стека указатель дескриптора сегмента программы, распределяется память в стековой памяти данных для вызываемой процедуры — в данном случае выделяется одно слово памяти для переменной M. Кроме того, производится присвоение значений дескрипторов косвенных адресов фактических параметров формальным параметрам процедуры в качестве начальных значений, соответствующие элементы помещаются в стек указателей сегментов программы и стек указателей активизируемых сегментов и обновляется содержимое дисплей-регистров. В результате этого начинается выполнение процедуры ZZZ.

При выполнении команды 54 дескриптор косвенного адреса, находящийся на вершине стека (помещенный туда командой 53), указывает на другой дескриптор косвенного адреса (слово, представляющее переменную M в слове 8 стековой памяти данных, которому было присвоено начальное значение как

Элементы таблицы символических имен

1	A	1	Float	
2	B	0	Fixed	
3	C	0	Fixed	
4		0	Fixed	4
5		0	Fixed	1
6	ZZZ	0	PROG SEG	20 800
7	M	0	Fixed	

Элементы таблицы областей действия переменных

1	1	6	0	0
2	7	1	1	1

Рис. 6.5. Таблица символических имен и таблица областей действия переменных для процедур XYZ и ZZZ.

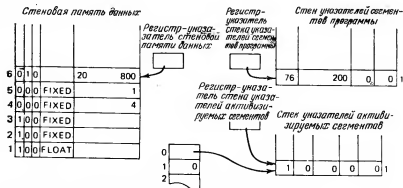


Рис. 6.6. Состояние машины при входе в процедуру XYZ.

параметру процедуры), который в свою очередь ссылается на переменную B. Реализация описываемых действий не вызывает затруднений, поскольку команда EVAL по определению способна «отслеживать» цепочку дескрипторов. Благодаря этому величина, находящаяся в конце цепочки, записывается на вершину стека.

Функции команды END почти полностью противоположны функциям команды ENTER. Однако, поскольку команда END используется также и для возврата из процедур-функций (в результате выполнения которых в вызывающую программу долж-

но передаваться определенное значение), предполагается, что на вершине стека находится вычисленное значение функции. После выполнения команды END локальная память для этой процедуры удаляется из стека и на вершину стека помещается возвращаемая величина. Так как вызов процедуры не сопровождается формированием такой величины, то генерируется команда UNDEF, по которой в стек загружается слово неопределенной величины. По команде 37 в вызывающем сегменте программы эта величина изымается из стека.

Команды 40—47 соответствуют оператору $A(B)=C$. При выполнении команд 40—43 дескриптор элемента массива помещается на вершину стека. Если значение переменной B оказывается за пределами граничных значений индекса массива, то при выполнении команды SUBS произойдет программное прерывание. Команда STORE производит запись значения переменной C (автоматически преобразуемое в форму с плавающей точкой) в элемент, адресуемый дескриптором элемента массива.

ДОСТОИНСТВА УЧЕБНОЙ ПЛ-МАШИНЫ

Чтобы оценить достоинства учебной ПЛ-машины по сравнению с традиционной ЭВМ, можно сравнить написанную на учебном языке ПЛ программу, приведенную в предыдущем разделе, с эквивалентной программой на языке ПЛ/1, компилируемой в Системе 370. Ниже дается текст этой программы:

```
(SUBSCRIPTRANGE): XYZ: PROCEDURE;
DECLARE A(*) FLOAT BINARY (21) CTL;
DECLARE (B,C) FIXED BINARY (31);
B=4;
ALLOCATE A(1:B);
A=1;
C=B+B*B;
CALL ZZZ(B);
A(B)=C;
  ZZZ: PROCEDURE (M);
    DECLARE M FIXED BINARY (31);
    C=M+M;
    END;
END
```

Программа, написанная на учебном языке ПЛ, состоит из 62 выполняемых команд, которые занимают в ПЛ-машине 96 байт памяти. Принимая во внимание таблицу символических имен и таблицу областей действия переменных, общий размер программы составляет 369 байт. При использовании в Системе 370 оптимизирующего транслятора языка ПЛ/1 фирмы IBM объектный код программы на языке ПЛ/1 занимает 564 байт памяти, а количество выполняемых команд равно 303. С учетом

выполнения подпрограмм суммарный объем памяти для программы в Системе 370 составляет 5912 байт.

Не вполне правильно сопоставлять 62 команды с 303 командами, так как в действительности при выполнении программы Системой 370 количество команд будет намного больше. Однако непосредственное сравнение количества команд не имеет существенного значения. Компилятор Системы 370 обеспечивает эффективные средства вызова подпрограмм и возврата из них, однако при этом в начале программы приходится выполнять большое число команд, выделяющих память для этих средств. Поскольку указанные команды выполняются за время прогона программы только один раз, при сравнении их можно не принимать во внимание.

Можно оспаривать утверждение о том, что проводимое сравнение необоснованно «смещено» не в пользу языка ПЛ/1, поскольку все массивы программы на учебном языке ПЛ должны размещаться динамически, а использование в приведенной выше программе на языке ПЛ/1 управляемой памяти¹⁾ и оператора ALLOCATE для массива A не является обязательным. Но если исключить из программы оператор ALLOCATE, удалить массив A из управляемой памяти и тем самым сделать недееспособной процедуру SUBSCRIPTRANGE, объектный код, генерируемый компилятором Системы 370, будет занимать 360 байт памяти, число выполняемых команд станет равным 112, а суммарный размер программы составит 5496 байт.

С точки зрения идей, изложенных в гл. 4, в учебной ПЛ-машине реализован принцип теговой памяти, которая используется для выполнения требуемых преобразований формы представления данных. Эта машина предоставляет пользователю набор команд, инвариантных к типу обрабатываемых данных, и располагает средствами обнаружения в программе ошибок нескольких типов. Машине «известно» понятие «массив данных»; в ней определены операции над массивами, индексирование элементов массивов; машина способна обнаруживать некорректно заданные индексы. Машина также располагает средствами управления подпрограммами, ориентированными на язык программирования блочной структуры; при этом машина способна обнаруживать несоответствие между типами аргументов в обращении к процедуре и типами параметров вызываемой процедуры.

Как следует из приведенного выше примера, параметр S учебной ПЛ-машины превосходит по величине подобные параметры для машин с традиционной архитектурой (при условии,

¹⁾ То есть указание атрибута CTL в операторе DECLARE. — Прим. перев.

что сравниваются программы и языки, для которых проектировалась исследуемая машина).

Однако параметр M , оценивающий в битах количество передаваемой информации, для этой машины оказывается ненамного выше, главным образом вследствие интенсивного обращения к памяти, занимаемой стеком операндов.

Объективно оценивая учебную ПЛ-машину, укажем несколько причин, по которым ее архитектуру (в том виде, как она представлена в настоящее время) следует признать непрактичной и даже нежелательной. Во-первых, эта машина представляет собой только модель вычислительного процесса, так что вопросы, связанные с операционной системой и организацией ввода-вывода, для нее не решены. Во-вторых, ее архитектура, ориентированная на язык программирования, слишком сильно «привязана» именно к учебному языку ПЛ. Например, компилирование программ, написанных на учебном языке ПЛ, производится непосредственно, однако компилирование программ на других языках таким же образом выполняться не может. Даже программы, написанные на языке ПЛ/1, наиболее близком к учебному языку ПЛ, на данной машине компилировать нельзя, поскольку многие конструкции языка ПЛ/1 (такие, как структуры, базированные переменные) не «поддерживаются» этой машиной. Кроме того, архитектура машины не позволяет использовать независимо компилируемые процедуры.

Менее существенным являются громоздкость и неэффективность описанного выше способа реализации операторов IF и DO с точки зрения как сложности компилирования, так и эффективности выполнения.

В этой связи можно заключить, что не следует вводить в архитектуру структуры, непосредственно отображающие все конструкции языка, особенно когда эти конструкции имеют преимущественно синтаксическую, а не семантическую природу. Несмотря на использование теговой памяти и набора команд, инвариантного к типу обрабатываемых данных, для машины свойственна избыточность памяти. Например, каждый элемент массива имеет идентичный, за исключением бита — признака неопределенной величины, 7-битовый тег, и поэтому представление логических величин является крайне неэффективным¹⁾.

УПРАЖНЕНИЯ

6.1. Какая информация находится в стеке данных перед выполнением команды ALLOC первого сегмента программы, приведенной на рис. 6.3?

6.2. Каково назначение восьми команд, следующих за командой ALLOC первого сегмента программы, приведенной на рис. 6.3?

¹⁾ Логическая переменная, принимающая значение 0 или 1, занимает полное слово памяти (39 бит). — *Прим. перев.*

6.3. Каким образом команда ENTER определяет вход в таблицу областей действия переменных (т. е. первый элемент таблицы, относящийся к данному сегменту программы) при вызове сегмента программы?

6.4. Как команда ENTER определяет соответствие количества фактических параметров процедуры количеству формальных параметров?

6.5. Каково назначение двух команд POP (команд с номерами 37 и 38) программы, приведенной на рис. 6.4?

6.6. Каким образом определяются элементы таблицы символических имен, представляющие собой формальные параметры, при инициализации параметров, производимой во время выполнения команды ENTER?

6.7. Для лучшего понимания архитектуры ЭВМ полезно мысленно выполнить компилирование исходной программы. Придумайте небольшую программу на учебном языке ПЛ и выполните в уме ее компилирование на учебной ПЛ-машине. (Все команды ПЛ-машины определены в гл. 7.)

ГЛАВА 7

НАБОР КОМАНД УЧЕБНОЙ ПЛ-МАШИНЫ

В этой главе приводится сводный перечень с соответствующей спецификацией команд ПЛ-машины. Спецификация команд заимствована из исходного описания машинного языка. Действия, выполняемые этой машиной при обнаружении ошибок в программе, не указываются, поскольку они были опущены и в первоначальном описании.

Для облегчения изучения команды разделены на пять групп. Команды доступа к данным и управления адресацией используются для выборки и записи адресов и величин данных в стековой памяти данных. Команды обработки данных выполняют арифметические и логические операции, а также операции над строками данных. Команды управления предназначены для управления последовательностью выполнения команд программы и реализации операторов IF и DO. Команды работы с процедурами используются для управления процедурами, процедурами-функциями и блоками BEGIN. В последнюю группу входят команды управления памятью массивов, с помощью которых распределяется память для массивов.

При описании команд показано их воздействие на содержимое стековой памяти данных, т. е. указаны величины, находящиеся в стеке до и после выполнения команд. При описании содержимого стека предполагается его горизонтальное расположение¹⁾: крайнее слева символическое имя обозначает информацию, расположенную на вершине стека, а символическое имя X представляет содержимое нижней части стека, не участвующей в операциях рассматриваемой команды.

В тех случаях, когда при описании команды указывается, что она использует дескриптор косвенного адреса, расположенный в стеке, соответствующее слово стека может содержать

¹⁾ Иначе говоря, содержимое стека представлено строкой символических имен, разделяемых запятыми: одно имя — одно слово стека. — *Прим. ред.*

дескриптор косвенного адреса некоторого операнда (одноуровневая косвенная адресация), дескриптор косвенного адреса другого дескриптора косвенного адреса и т. д., указывая, наконец, адрес некоторого операнда (многоуровневая косвенная адресация) или непосредственно операнд.

КОМАНДЫ ДОСТУПА К ДАННЫМ И УПРАВЛЕНИЯ АДРЕСАЦИЕЙ

Имя команды. SNAME

Длина команды. 16 бит.

Выполняемая операция. Загрузка в стек дескриптора косвенного адреса адресуемого слова.

Содержимое информационного поля команды. Адрес лексического уровня. Первые 4 бит — номер лексического уровня, последние 4 бит — значение индекса.

Исходное содержимое стека. X

Конечное содержимое стека. VAR, X

Операнды. VAR — дескриптор косвенного адреса, формируемый из адреса лексического уровня. Если адресуемое слово содержит дескриптор косвенного адреса, то VAR принимает его значение.

Имя команды. LNAME

Длина команды. 24 бит.

Выполняемая операция. Загрузка в стек дескриптора косвенного адреса адресуемого слова.

Содержимое информационного поля команды. Адрес лексического уровня. Первые 4 бит представляют собой номер лексического уровня, последние 12 бит — значение индекса.

Исходное содержимое стека. X

Конечное содержимое стека. VAR, X

Операнды. VAR — дескриптор косвенного адреса, формируемый из адреса лексического уровня. Если адресуемое слово содержит дескриптор косвенного адреса, то VAR принимает его значение.

Имя команды. EVAL

Длина команды. 8 бит.

Выполняемая операция. Замена дескриптора, находящегося на вершине стека, содержимым адресуемого слова.

Исходное содержимое стека. VAR1, X

Конечное содержимое стека. VAR2, X

Операнды. Если переменная VAR1 не является дескриптором косвенного адреса или дескриптором элемента массива, то VAR2=VAR1, иначе значение VAR2 равно содержимому адре-

суемого слова. Переменная VAR1 не может быть дескриптором косвенного адреса или дескриптором элемента массива; для указания местоположения требуемых данных используется цепочка дескрипторов, ссылающихся друг на друга. Если значение переменной VAR2 не определено, то выдается сообщение об ошибке.

Имя команды. STORE

Длина команды. 8 бйт.

Выполняемая операция. Запись содержимого слова, находящегося на вершине стека, в память по адресу, указываемому вторым словом стека.

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. VAR1, X

Операнды. Переменная VAR2 — дескриптор косвенного адреса, дескриптор элемента массива или указатель массива. Переменная VAR1 записывается в адресуемую область памяти. Если VAR1 — указатель массива, а адресуемая ячейка памяти содержит указатель массива, то второй массив заполняется содержимым первого. Если переменная VAR1 не является указателем массива, то ее значение записывается во все элементы второго массива. Если тип переменной VAR1 отличен от типа данных, указываемого тегом адресуемой области памяти, то автоматически выполняется требуемое преобразование данных.

Примечание. Правила преобразования не были точно определены в первоначальном описании, но, по всей видимости, они те же, что и соответствующие правила языка ПЛ/1.

Имя команды. SUBS

Длина команды. 16 бит.

Выполняемая операция. Формирование дескриптора элемента адресуемого массива или части массива (секции массива).

Содержимое информационного поля команды. N — число индексов.

Исходное содержимое стека. VAR1, ..., VARN, VARP, X

Конечное содержимое стека. VAR Q, X

Операнды. VAR1 — VARN — переменные, каждая из которых является индексом элемента массива и может быть представлена логической величиной, числом с фиксированной или плавающей точкой, либо знаком звездочки. При использовании логических величин или чисел с плавающей точкой осуществляется их преобразование в числа с фиксированной точкой. Переменная VARP — это указатель массива или дескриптор косвенного адреса (либо начало цепочки дескрипторов косвенного адреса) указателя массива.

Если ни один из индексов не задан в виде звездочки, то пере-

менная VARQ — дескриптор адресуемого элемента массива. Если для задания одного или нескольких индексов использована звездочка, то в памяти дескрипторов массивов формируется дескриптор определенной части массива, а VARQ будет представлять собой указатель этого дескриптора. Если значения индексов оказываются за их пределами для соответствующего массива или N не равно размерности массива, выдается сообщение об ошибке.

Имя команды. POP

Длина команды. 8 бит.

Выполняемая операция. Удаление слова, находящегося на вершине стека.

Исходное содержимое стека. VAR, X

Конечное содержимое стека. X

Имя команды. SWAP

Длина команды. 8 бит.

Выполняемая операция. Обмен содержимым двух слов, находящихся в верхней части стека.

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. VAR2, VAR1, X

Имя команды. UNDEF

Длина команды. 8 бит.

Выполняемая операция. Загрузка в стек слова, содержащего неопределенную величину.

Исходное содержимое стека. X

Конечное содержимое стека. VAR, X

Операнды. Значение VAR не определено (бит и тега слова равен 1).

КОМАНДЫ ОБРАБОТКИ ДАННЫХ

Имя команды. ADD, SUB, MUL, DIV, PWR

Длина команды. 8 бит.

Выполняемая операция. Выполнение арифметических операций над содержимым двух слов, находящихся в верхней части стека.

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. VAR3, X

Операнды. Значения переменных VAR1 и VAR2 должны быть числами с фиксированной или плавающей точкой. Если одна переменная представлена в форме с плавающей точкой, то вторая автоматически преобразуется в такую же форму. Операция выполняется согласно следующему правилу:

$$\text{VAR3} = \text{VAR2} . \text{op. VAR1}$$

где *op* — код операции. Если переменные *VAR1* и *VAR2* — указатели массивов, то операция выполняется над соответствующими элементами этих массивов. В результате создается новый массив, причем переменная *VAR3* служит его указателем. Если переменная *VAR1(VAR2)* — указатель некоторого массива, а переменная *VAR2(VAR1)* — скалярная величина, то над каждым элементом массива выполняется арифметическая операция как над скаляром. В результате создается новый массив с переменной *VAR3* в качестве указателя этого массива.

Замечание. Преобразование формата данных исходных операндов не влияет на форму представления операнда в памяти. Это преобразование производится только во время выполнения команды.

Имя команды. CAT

Длина команды. 8 бит.

Выполняемая операция. Формирование строки символов путем конкатенации (сцепления) двух строк.

Исходное содержимое стека. *VAR1, VAR2, X*

Конечное содержимое стека. *VAR3, X*

Операнды. Переменные *VAR1* и *VAR2* — дескрипторы логических величин, чисел с фиксированной или плавающей точкой либо строки символов. Если указанные переменные не являются дескрипторами строк символов, то адресуемые данные преобразуются в строки символов. В памяти сегментов программ и строк данных формируется новая строка символов, состоящая из строки, задаваемой переменной *VAR2*, за которой следует строка, задаваемая переменной *VAR1*. Переменная *VAR3* — дескриптор строки символов. Как и в предыдущих командах, если переменная *VAR1* и (или) *VAR2* являются указателями массивов, то операция выполняется над элементами массивов, причем переменная *VAR3* становится указателем нового массива.

Имя команды. LT, GT, LE, GE, EQ, NE

Длина команды. 8 бит.

Выполняемая операция. Выполнение операций сравнения над содержимым двух слов, находящихся в верхней части стека.

Исходное содержимое стека. *VAR1, VAR2, X*

Конечное содержимое стека. *VAR3, X*

Операнды. Сравнение выполняется согласно следующему правилу:

$VAR3 = VAR2.$ *op.* *VAR1*

где *op* — код операции. Если переменные *VAR1* и *VAR2* — данные различных типов, то данные, тип которых принадлежит бо-

лее низкому уровню иерархии типов, преобразуются в форму данных более высокого уровня. Иерархия типов данных в порядке возрастания их уровней имеет следующий вид: логические величины, числа с фиксированной точкой, числа с плавающей точкой, символьные данные. VAR3 — логическая переменная, принимающая одно из двух значений: TRUE (ИСТИННО) или FALSE (ЛОЖНО). Если переменные VAR1 и (или) VAR2 — указатели массивов, то выполняется сравнение элементов двух массивов или элементов массива со скалярной величиной; в этом случае VAR3 — указатель нового массива логических величин.

Имя команды. AND, OR

Длина команды. 8 бит.

Выполняемая операция. Выполнение логической операции над содержимым двух слов, находящихся в верхней части стека.

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. VAR3, X

Операнды. Значения переменных VAR1 и VAR2 преобразуются, если это требуется, в логические величины, после чего выполняется соответствующая операция и переменная VAR3 принимает то или иное логическое значение. Если переменные VAR1 и (или) VAR2 — указатели массивов, то операция выполняется над элементами массивов, а переменная VAR3 — указатель нового массива.

Имя команды. PLUS, NEG

Длина команды. 8 бит.

Выполняемая операция. Сохранение неизменным (по команде PLUS) или изменение на противоположный (по команде NEG) знака операнда.

Исходное содержимое стека. VAR1, X

Конечное содержимое стека. VAR2, X

Операнды. Переменная VAR1 — логическая величина, число с фиксированной или плавающей точкой. Если это логическая величина, то она преобразуется в число с фиксированной точкой. В результате выполнения команды PLUS имеет место соотношение $VAR2 = VAR1$, а реализация команды NEG приводит к равенству $VAR2 = -VAR1$. Если переменная VAR1 служит указателем массива, то формируется новый массив, при этом переменная VAR2 является указателем нового массива.

Имя команды. NOT

Длина команды. 8 бит.

Выполняемая операция. Формирование логического дополнения операнда.

Исходное содержимое стека. VAR1, X

Конечное содержимое стека. VAR2, X

Операнды. Если необходимо, то переменная VAR1 преобразуется в логическую величину; в результате выполнения команды логическое дополнение этой величины присваивается переменной VAR2. Если переменная VAR1 — указатель массива, то формируется новый массив, указателем которого является VAR2.

Имя команды. BIT

Длина команды. 8 бит.

Выполняемая операция. Преобразование операнда в логическую величину.

Исходное содержимое стека. VAR1, X

Конечное содержимое стека. VAR2, X

Операнды. Переменная VAR1 — логическая величина либо число с фиксированной или плавающей точкой. Если значение переменной VAR1 равно нулю, то значение логической переменной VAR2 ложно, иначе — истинно.

КОМАНДЫ УПРАВЛЕНИЯ

Имя команды. CALL

Длина команды. 8 бит.

Выполняемая операция. Простановка выполнения текущего сегмента программы и инициирование выполнения другого сегмента.

Исходное содержимое стека. VAR, X

Конечное содержимое стека. X

Операнды. Переменная VAR — дескриптор сегмента программы. В стеке сегментов программ формируется адрес точки входа в новый сегмент.

Примечание. Эта команда — не обращение к процедуре, поскольку не предусмотрена передача параметров и не происходит изменения лексического уровня. Команда используется для реализации операторов IF и DO. Команда CALL по производимому действию подобна оператору PERFORM языка КОБОЛ.

Имя команды. DOSTORE

Длина команды. 8 бит.

Выполняемая операция. Запись содержимого слова, находящегося на вершине стека, в область памяти, адресуемую вторым словом стека.

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. VAR2, X

Операнды. Переменная VAR2 — дескриптор косвенного адреса или дескриптор элемента массива.

Примечание. Команда DOSTORE предназначена для инициализации итеративного цикла DO, причем переменная VAR1 — начальное значение параметра цикла, а переменная VAR2 — его текущее значение. Команда DOSTORE выполняет ту же операцию, что и команда STORE, но имеет более ограниченный диапазон действия. При необходимости переменная VAR1 преобразуется в форму, соответствующую типу переменной VAR2.

Имя команды. DOTEST

Длина команды. 8 бит.

Выполняемая операция. Сравнение двух величин с целью определения, равны ли они, а если нет, то которое из них больше (меньше).

Исходное содержимое стека. VAR1, VAR2, VAR3, X

Конечное содержимое стека. VAR4, VAR1, VAR2, VAR3, X

Операнды. Переменная VAR3 — дескриптор косвенного адреса числовой величины. Переменные VAR1 и VAR2 — числовые величины. VAR4 — логическая переменная, значение которой истинно, если 1) значение переменной VAR1 больше нуля, а содержимое слова, адресуемого переменной VAR3, меньше или равно значению переменной VAR2; 2) значение переменной VAR1 меньше нуля, а содержимое слова, адресуемого переменной VAR3, больше или равно значению переменной VAR2.

Примечание. Команда DOTEST предназначена для организации итеративных циклов DO, причем переменная VAR1 — приращение, переменная VAR2 — предельное значение, а переменная VAR3 — текущее значение параметра цикла.

Имя команды. CRET

Длина команды. 8 бит.

Выполняемая операция. Возврат в предшествующий сегмент программы, если логическая величина, находящаяся на вершине стека, имеет значение «ложно».

Исходное содержимое стека. VAR, X

Конечное содержимое стека. X

Операнды. VAR — логическая переменная. Если ее значение равно 1 (истинно), то выполняется следующая команда данного сегмента программы. Если же значение VAR равно 0 (ложно), то выполнение текущего сегмента программы прекращается и возобновляется выполнение предшествующего сегмента.

Примечание. Команда CRET предназначена для выполнения циклов DO WHILE и итеративных циклов DO. В итеративном цикле эта команда следует за командой DOTEST.

Имя команды. DOINCR

Длина команды. 8 бит.

Выполняемая операция. Увеличение содержимого третьего от вершины стека слова на величину, находящуюся на вершине стека.

Исходное содержимое стека. VAR1, VAR2, VAR3, X

Конечное содержимое стека. VAR1, VAR2, VAR3, X

Операнды. Переменная VAR3 — дескриптор косвенного адреса, по которому находятся числовые данные. Переменная VAR1 — числовая величина. Переменная VAR1 добавляется к числовым данным, адресуемым VAR3.

Примечание. См. примечание к описанию команды DOTEST.

Имя команды. CYCLE

Длина команды. 8 бит.

Выполняемая операция. Возврат управления первой команде данного сегмента программы.

Исходное содержимое стека. X

Конечное содержимое стека. X

Примечание. Команда CYCLE предназначена для организации циклов DO.

Имя команды. GO TO

Длина команды. 8 бит.

Выполняемая операция. Передача управления команде, указываемой дескриптором метки.

Исходное содержимое стека. VAR, X

Конечное содержимое стека. X

Операнды. Переменная VAR — дескриптор косвенного адреса дескриптора метки-константы или метки-переменной. Если команда, которой передается управление, находится в данный момент в неактивизированном сегменте программы, то выполнение всех активизированных сегментов завершается, и, если это необходимо, производится обновление содержимого дисплей-регистров.

Имя команды. HALT

Длина команды. 8 бит.

Выполняемая операция. Завершение выполнения программы.

Исходное содержимое стека. X

Конечное содержимое стека. 0

Имя команды. LINE

Длина команды. 16 бит.

Выполняемая операция. Загрузка числа N в регистр номера оператора исходной программы.

Содержимое информационного поля команды: N (слово длиной 8 бит).

Исходное содержимое стека. X

Конечное содержимое стека. X

КОМАНДЫ РАБОТЫ С ПРОЦЕДУРАМИ

Имя команды. ENTER

Длина команды. 16 бит.

Выполняемая операция. Прекращение выполнения текущего сегмента программы и начало выполнения другого сегмента, возможно находящегося на другом лексическом уровне.

Содержимое информационного поля команды. N — число передаваемых параметров (слово длиной 8 бит).

Исходное содержимое стека. VARP, VAR1, ..., VARN, X

Конечное содержимое стека. Указатель памяти для вызываемой процедуры, VAR1, ..., VARN, X

Операнды. Переменная VARP — дескриптор косвенного адреса дескриптора сегмента программы. VAR1 — VARN — дескрипторы косвенных адресов фактических параметров. Команда ENTER проверяет совпадения числа и типов фактических параметров с числом и типами формальных параметров процедуры. Она распределяет память в стеке данных для переменных нового сегмента программы, присваивает формальным параметрам процедуры в качестве начальных значений значения, определяемые дескрипторами косвенных адресов фактических параметров, помещает соответствующие элементы в стек указателей активизируемых сегментов программы и стек указателей сегментов программы, а также обновляет содержимое дисплей-регистров. Если фактический параметр и соответствующий формальный параметр процедуры отличаются типами данных, которые они представляют, и при этом возможно корректное преобразование формата данных, то дескриптор косвенного адреса фактического параметра замещается дескриптором преобразованного надлежащим образом фактического параметра.

Примечание. Команда ENTER используется для вызова процедур и процедур-функций и организации входа в блоки BEGIN

Имя команды. SCOPEID

Длина команды. 16 бит.

Выполняемая операция. При выполнении этой команды не производится никаких операций.

Содержимое информационного поля команды. M — адрес элемента таблицы областей действия переменных, соответствующего данному сегменту программы (слово длиной 8 бит).

Исходное содержимое стека. X

Конечное содержимое стека. X

Примечание. Команда SCOPEID должна быть первой командой любого сегмента программы, вызываемого по команде ENTER. Величина M используется командой ENTER для определения местоположения соответствующего элемента в памяти таблицы областей действий переменных.

Имя команды. END

Выполняемая операция. Завершение выполнения сегмента программы, вызванного по команде ENTER, и возобновление выполнения вызывающего сегмента.

Содержимое информационного поля команды. M — адрес элемента таблицы областей действия переменных, соответствующего данному сегменту программы (слово длиной 8 бит).

Исходное содержимое стека. VAR, X

Конечное содержимое стека. VAR, X1

Операнды. Переменная VAR является величиной, поступающей обратно в вызывающий сегмент программы. X1 — разность значений содержимого стека в начале выполнения соответствующей команды ENTER и переменной VARP (дескриптора косвенного адреса дескриптора сегмента программы). Команда END восстанавливает предшествующее состояние стека указателей активизируемых сегментов программы и стека указателей сегментов программы, а также обновляет содержимое дисплей-регистров.

Имя команды. PARAM.

Длина команды. 16 бит.

Выполняемая операция. Приостановка выполнения текущего сегмента программы и инициирование выполнения процедуры, если операнд команды указывает процедуру. Если операнд не указывает процедуру, то никаких операций не производится и содержимое стека сохраняется неизменным.

Содержимое информационного поля команды. P — число (слово длиной 8 бит).

Исходное содержимое стека. VAR1, VAR2, X

Конечное содержимое стека. Указатель памяти для вызываемой процедуры, VAR2, X

Операнды. Переменная VAR1 — дескриптор косвенного адреса фактического параметра. Число P указывает, что VAR1 представляет P-й параметр для процедуры, указываемой VAR2. (Переменная VAR2 — дескриптор косвенного адреса дескриптора сегмента программы.) Если фактический параметр — имя процедуры-функции без параметров, замещаемое в результате ее работы некоторой величиной (т. е. если VAR1 указывает

дескриптор сегмента программы), то в сегмент, определяемый VARI, управление передается так, как если бы команда ENTER была выполнена.

Примечание. Команда PARAM используется для определения того, является ли формальный параметр процедуры, соответствующий передаваемому фактическому параметру, именем процедуры, или нет. В последнем случае значение фактического параметра должно быть определено. Компилятор генерирует команды PARAM перед каждой командой ENTER.

Имя команды. LFUNC

Длина команды. 16 бит.

Выполняемая операция. Приостановка выполнения текущего сегмента программы и начало выполнения другого сегмента, представляющего псевдопеременную.

Содержимое информационного поля команды. N — число фактических параметров (слово длиной 8 бит).

Исходное содержимое стека. VARR, VARF, VARI, ..., VARN, X

Конечное содержимое стека. Не определено.

Операнды. Переменная VARR — дескриптор косвенного адреса величин, преобразуемой в строку символов. Переменная VARF — дескриптор косвенного адреса числа с фиксированной точкой, для которого вызывается соответствующая встроенная функция. Переменные VARI — VARN — дескрипторы адресов фактических параметров.

Примечание. Команда LFUNC используется для представления псевдопеременных OUTPUT и SUBSTR в левой части операторов присваивания. Переменная VARR представляет значение правой части оператора присваивания. Действие команды LFUNC подобно обращению к супервизору.

КОМАНДЫ УПРАВЛЕНИЯ ПАМЯТЬЮ МАССИВОВ

Имя команды. ALLOC

Длина команды. 16 бит.

Выполняемая операция. Распределение памяти для массива данных.

Содержимое информационного поля команды. D — число, обозначающее размерность массива (слово длиной 8 бит).

Исходное содержимое стека. VARA, VARL1, VARU1, ..., VARLD, VARUD, X

Конечное содержимое стека. X

Операнды. Переменная VARA — дескриптор косвенного адреса указателя массива. Переменные VARL1 — VARUD — числа с фиксированной точкой, представляющие собой минимальное и

максимальные граничные значения индексов для каждого измерения массива. Команда **ALLOC** распределяет память для элементов массива (в верхней части стека данных), формирует дескриптор в памяти дескрипторов массивов и записывает адрес дескриптора в указатель массива. Если существовала предыдущая версия массива (т. е. если указатель массива не является неопределенной величиной), то дескриптор нового массива указывает местоположение дескриптора предшествующей версии массива.

Имя команды. **FREE**

Длина команды. 8 бит.

Выполняемая операция. Освобождение памяти, занимаемой массивом или одной из версий массива.

Исходное содержимое стека. **VARA, X**

Конечное содержимое стека. **VARA, X**

Операнды. **VARA** — дескриптор косвенного адреса указателя массива. Освобождается память, занимаемая элементами массива, а также относящимся к массиву дескриптором (в памяти дескрипторов массивов). Если существует ранее созданная версия массива (дескриптор массива содержит ссылку на дескриптор другого массива), то указатель массива модифицируется так, чтобы указывать на дескриптор следующего массива; если такой версии массива нет, то указатель массива помечается как неопределенный.

ЧАСТЬ III

АРХИТЕКТУРА МАШИН ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

ГЛАВА 8

АРХИТЕКТУРА СИСТЕМЫ SYMBOL

В качестве второго объекта анализа достижений в совершенствовании архитектуры ЭВМ выбрана функционирующая вычислительная система SYMBOL [1—28], разработанная фирмой Fairchild Camera and Instrument. Единственный сконструированный образец этой системы изучался в течение нескольких лет в Университете шт. Айова, после чего от использования этой системы отказались.

Система SYMBOL была разработана в результате выполнения большого исследовательского проекта по анализу и пересмотру традиционного подхода к разделению функций, реализуемых вычислительной системой, между аппаратными средствами и программным обеспечением. Цели проводимого исследования заключались в следующем:

1. Спроектировать вычислительную систему с существенно повышенной по сравнению с обычными системами производительностью и меньшей стоимостью вычислений за счет непосредственной аппаратной реализации языка программирования высокого уровня, виртуальной памяти и режима разделения времени операционной системы.

2. Создать систему, ориентированную не столько на традиционную обработку числовых данных, сколько на работу с нечисловой информацией и данными сложной структуры.

3. Разработать новые принципы проектирования и методы конструирования аппаратных средств (одним из результатов исследований в указанном направлении явилась разработка корпусов интегральных схем с двурядным расположением выводов; эти корпуса пользуются и в настоящее время высоким спросом).

4. Продемонстрировать реализуемость сформулированных выше целей путем создания полноценной и работоспособной системы.

Придерживаясь классификации, приведенной в гл. 3, архитектуру системы SYMBOL можно отнести к архитектуре типа Б машины языков высокого уровня. Набор команд рассматривае-

мой вычислительной системы — это префиксная запись операторов языка программирования SPL (SYMBOL Programming Language — язык программирования SYMBOL), причем компилирование выполняется в значительной мере аппаратными, а не программными средствами.

Уже первая из сформулированных выше целей создания системы SYMBOL довольно необычна. Словосочетание «непосредственная аппаратная реализация» означает, что система фактически не содержит программного обеспечения. Точнее говоря, определенные средства программного обеспечения имеются, но выполняют только несколько второстепенных функций, так что система может работать без них. Кроме того, в системе отсутствуют микропрограммы, все ее функции реализуются набором последовательностных логических схем. Например, транслятор — это аппаратно выполненная последовательностная схема; алгоритмы адресации памяти и замещения страниц памяти реализуются с помощью последовательностных логических схем; команды, вводимые пользователем с терминала, обрабатываются последовательностными логическими схемами и т. д. Хотя описание может показаться самой необычной характеристикой системы SYMBOL, эту характеристику не относят к главным, поскольку она оказывается одним из недостатков системы, так как указанные здесь функции системы могли быть обеспечены средствами микропрограммирования. Следует заметить, что в начале работы над проектом, т. е. в середине 60-х годов, микропрограмное управление еще не было широко распространенным средством реализации функций вычислительной системы.

Для пользователя система SYMBOL представляется терминально-ориентированной системой с виртуальной памятью, работающей в режиме разделения времени и поддерживающей язык SPL. Внутренняя организация и машинный язык системы не доступны пользователю. Он выполняет отладку программы, пользуясь языком программирования, и никогда не имеет дела с традиционными машинными адресами или дампом оперативной памяти.

КОНФИГУРАЦИЯ СИСТЕМЫ

По внутренней организации система SYMBOL является мультипроцессорной системой, которая, однако, отличается от традиционных мультипроцессорных систем двумя уникальными свойствами. Во-первых, пользователь находится в абсолютном неведении относительно того, что эта система мультипроцессорная. Во-вторых, каждый процессор предназначен для выполнения определенной функции: один — для компилирования программ, написанных на языке SPL; другой — для диспетчериза-

ции и управления обменом страницами (их «перелистыванием»), третий — для обработки команд, поступающих с терминала; четвертый — для управления памятью; пятый — для выполнения программ пользователя и т. д.

Процессоры, входящие в состав системы, и их взаимосвязи показаны на рис. 8.1. Число, находящееся в правом нижнем

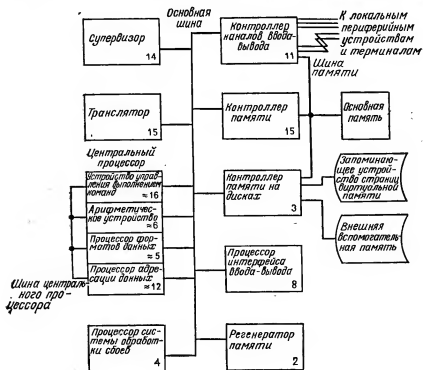


Рис. 8.1. Функциональная схема мультипроцессорной системы SYMBOL.

углу каждого блока, — мера относительной сложности соответствующего процессора; оно обозначает количество монтажных плат в процессоре. Каждая плата содержит ~200 интегральных схем или ~800 вентиляей.

Функции, реализуемые каждым процессором, рассматриваются в гл. 9 и 10, однако их краткое описание целесообразно привести здесь. Для каждого процессора имеется своя очередь запросов, которые должны быть обслужены, и все процессоры могут работать одновременно. Программа пользователя переходит от одного процессора к другому по мере того, как возник-

кает необходимость в реализации различных функций, связанных с выполнением программы. Для конкретной программы пользователя в данный момент всегда работает только один процессор. Таким образом, с точки зрения отдельного пользователя система представляется однопроцессорной.

Контроллер каналов ввода-вывода является промежуточным звеном между 32 каналами ввода-вывода (каждый из которых представляет собой удаленный терминал или локальное периферийное устройство ввода-вывода) и связанных с этими каналами буферами в основной памяти. *Процессор интерфейса ввода-вывода* также выполняет функции, относящиеся к организации ввода-вывода данных. Когда буфер ввода заполнен, этот процессор передает содержимое буфера в соответствующую область виртуальной памяти, выделенную пользователю. Процессор интерфейса ввода-вывода работает так же, как и система управления вводом-выводом традиционной операционной системы: при выполнении команды ввода или вывода, содержащейся в программе пользователя, управление выполнением этой операции ввода-вывода осуществляется данным процессором. Процессор интерфейса ввода-вывода реализует также функции подсистемы обработки директив пользователя традиционной системы разделения времени. Например, если пользователь со своего терминала редактирует текст исходной программы, то команды редактирования обрабатываются данным процессором, благодаря чему другие процессоры оказывают незагруженными и могут выполнять запросы других пользователей.

Транслятор представляет собой отдельный процессор, единственной функцией которого является компилирование программы. Эта функция заключается в выборке программ на языке SPL из входной очереди транслятора и их компилировании в объектные программы.

Единственной функцией *центрального процессора* является выполнение объектных программ. Как показано на рис. 8.1, центральный процессор в действительности состоит из четырех отдельных подпроцессоров. Эти процессоры работают последовательно, так что в любой момент времени центральный процессор выполняет только одну программу.

Четыре процессора являются обслуживающими: они не выполняют самостоятельно определенных функций системы, а обрабатывают запросы других процессоров. Так, *контроллер памяти* обслуживает запросы других процессоров на чтение информации из памяти или запись ее в память. Это устройство позволяет другим процессорам «представлять» память с высокой степенью абстракции: для других процессоров память — это бесконечное число списков бесконечной длины. Такое представление памяти и соответствующих процессов управления ею яв-

ляется основной особенностью системы SYMBOL и обсуждается более детально в гл. 10.

Вторым обслуживающим процессором является *регенератор памяти*. Вследствие того что контроллер отображает логическую структуру памяти (списковые структуры) в виртуальную память, а виртуальной памяти ставит в соответствие адекватные разделы основной памяти, обслуживание запросов процессора на освобождение памяти требует значительного времени. Поэтому вместо того, чтобы обрабатывать такие запросы последовательно, связывая между собой запрашивающий процессор и контроллер памяти на чрезмерно большой интервал времени, контроллер памяти просто помечает соответствующую область как недоступную каким-либо запросам и помещает ее в очередь на освобождение к регенератору памяти.

Третий процессор — это *супервизор системы* (или просто супервизор), выполняющий функции, аналогичные функциям диспетчеризации и управления страничным обменом традиционной операционной системы. Супервизор управляет очередями к другим процессорам и планирует операции манипулирования страницами, когда один из процессоров, производя обращение к контроллеру памяти, сталкивается с проблемой отсутствия требуемой страницы в основной памяти.

Последним обслуживающим процессором является *контроллер памяти на магнитных дисках*. Это устройство функционирует как управляемый очередью канал ввода-вывода; оно обрабатывает запросы, находящиеся в его очереди, и осуществляет передачу информации между основной памятью и внешними запоминающими устройствами.

В системе имеется также *процессор обработки сбоев*. При нормальном функционировании системы этот процессор в работе не участвует. Он используется для отладки, а также прослеживания особых ситуаций, возникающих на основной шине, соединяющей все процессоры. Шина предназначена для передачи следующей информации:

- 1) данных между любым процессором и контроллером памяти;
- 2) управляющей информации между любым процессором и супервизором;
- 3) данных между четырьмя компонентами центрального процессора.

При посылке запросов к контроллеру памяти процессоры используют основную шину поочередно согласно установленной для них системе приоритетов.

Поскольку память является наиболее важным ресурсом системы, установленный приоритет дает представление об относительной значимости самих процессоров. Иерархия процессо-

ров в порядке убывания их приоритетов доступа к памяти имеет следующий вид: супервизор, процессор интерфейса ввода-вывода, центральный процессор, транслятор, регенератор памяти. Контроллер каналов ввода-вывода и контроллер памяти на дисках не включены в этот список, поскольку они не используют контроллер памяти; данные устройства подключены непосредственно к шине памяти (рис. 8.1). Третья шина используется в центральном процессоре для передачи управляющей информации между четырьмя его компонентами.

ПРОХОЖДЕНИЕ ПОТОКА ЗАДАНИЙ ЧЕРЕЗ СИСТЕМУ

Динамику работы системы легче всего понять путем изучения прохождения потока заданий пользователей от процессора к процессору. Для этого сделаем ряд простейших допущений. Предположим, что вызов новых страниц виртуальной памяти не производится, т. е. все необходимые данные находятся в основной памяти. Известно, что каждый процессор обрабатывает помещенные в его очередь запросы в течение некоторого предельно допустимого интервала времени. Мы будем полагать, что время полной обработки запроса не выходит за пределы этого интервала. «Перелистывание» страниц виртуальной памяти и временная коммутация обслуживания запросов (путем разбиения времени обработки одного запроса на отдельные интервалы) описываются в гл. 10.

Терминал пользователя системы SYMBOL имеет в добавление к обычной буквенно-цифровой клавиатуре набор функциональных клавиш для ввода команд управления системой. Примерами таких клавиш являются следующие: LOAD, RUN, PAUSE, CONTINUE, SEARCH, REMOVE, FORWARD, BACKWARD, DISPLAY, причем последние пять клавиш предназначены для выполнения функций редактирования исходного текста на экране дисплея. Первоначально будем полагать, что пользователь работает в режиме загрузки, ввода и редактирования текста. При этом используются только четыре процессора: супервизор, контроллер каналов ввода-вывода, процессор интерфейса ввода-вывода и контроллер памяти. Весьма вероятно, что эти четыре процессора в то же время выполняют запросы других пользователей, а транслятор и центральный процессор работают независимо, обрабатывая задания других пользователей.

Если пользователь вводит со своего терминала строку текста (например, оператор написанной им программы), то контроллер каналов ввода-вывода принимает каждый символ и помещает его в буфер строк, относящийся к этому терминалу и расположенный в зарезервированной для него области основной

памяти. После заполнения буфера контроллер каналов ввода-вывода посылает супервизору сигнал «Буфер заполнен» и переключает данный терминал на запасной буфер. Супервизор помещает запрос в очередь к процессору интерфейса ввода-вывода, указывая ему на необходимость передачи содержимого заполненного буфера во временную рабочую область пользователя. «Встретив» этот запрос в своей входной очереди, процессор интерфейса ввода-вывода пересылает содержимое буфера в рабочую область, обращаясь для этой цели с надлежащими запросами к контроллеру памяти.

Если пользователь нажимает одну из командных клавиш (например, клавишу BACKWARD для перемещения указателя текущей строки в рабочей области назад на одну строку), контроллер каналов ввода-вывода принимает соответствующий управляющий символ. Выяснив, что принят управляющий символ, контроллер не помещает его в буфер, а посылает супервизору. Если управляющий символ запрашивает выполнение операции редактирования текста (например, BACKWARD), то супервизор помещает запрос в очередь процессора интерфейса ввода-вывода. Когда наступит очередь обработки данного запроса, указанный процессор выполнит требуемую операцию редактирования, обращаясь для этой цели к контроллеру памяти.

Другие действия производятся, когда пользователь нажимает клавишу RUN, инициирующую трансляцию и выполнение программы, находящейся во временной рабочей области пользователя. Как и ранее, контроллер каналов ввода-вывода распознает управляющий символ и посылает его супервизору. Последний создает входной элемент в очереди запросов транслятора. В дальнейшем транслятор выберет данный элемент из своей очереди и начнет компилирование программы.

Транслятор выполняет функции как компилирования, так и редактирования программы. Если в программе встречается так называемая неразрешенная ссылка на процедуру, то транслятор выбирает эту процедуру из соответствующей библиотеки программ¹⁾. (Библиотеки программ входят в состав файлов системы, которые отображаются в виртуальной памяти, образуя таким образом одноуровневую память.) Завершив компилирование программы, транслятор, производя обращение с соответствующими запросами к контроллеру памяти, создает в памяти два списка: объектный код программы и одну или более таблиц символических имен.

Транслятор работает настолько быстро, что было принято решение хранить все программы (речь идет о программах в

¹⁾ «Разрешение» подобной ссылки как раз и заключается в описываемых действиях транслятора. — *Прим. перев.*

библиотеках) в исходной форме. Если не учитывать операций ввода-вывода и «перелыстывания» страниц виртуальной памяти, то средняя скорость компилирования равна 75 000 оператор/мин. В оптимизированной версии системы SYMBOL, которая была спроектирована, но так никогда и не была сконструирована, транслятор должен был обеспечить компилирование в среднем 300 000 оператор/мин [2].

При завершении компилирования программы транслятор извещает об этом супервизор системы. Супервизор помещает соответствующий запрос во входную очередь центрального процессора. В соответствии с очередностью этого запроса центральный процессор начинает выполнение программы. Центральный процессор использует контроллер памяти для доступа к объектному коду программы и таблицам символических имен, выделения памяти для переменных программы и изменения значений этих переменных. Если центральный процессор встречает в объектном коде команду ввода-вывода, то он обращается к супервизору, который помещает запрос на ввод-вывод в очередь процессора интерфейса ввода-вывода.

Может показаться, что в системе SYMBOL все межпроцессорные связи должны приводить к значительному расходу машинного времени. Хорошим способом оценки данного фактора и сравнения системы SYMBOL с традиционной системой разделения времени является анализ времени, требуемого для компилирования и выполнения программы, состоящей из одного оператора и не совершающей никакой обработки. Для случая, когда с системой работает единственный пользователь, интервал времени от приема команды RUN до завершения выполнения программы составляет 315 мкс. Применение такого же критерия для оценки традиционной системы показывает, что здесь временной интервал более 315 мкс требуется только для анализа команды на выполнение компилирования, не говоря уже о времени, затрачиваемом на запуск транслятора. Подобная оценка производительности системы SYMBOL (315 мкс) представляется еще более удивительной, если принять во внимание, что фактором, ограничивающим быстродействие системы, является использование относительно медленнодействующей ферритовой памяти с временем цикла обращения к 64-разрядному слову, равным 4 мкс.

ЯЗЫК ПРОГРАММИРОВАНИЯ СИСТЕМЫ SYMBOL

Язык программирования системы SYMBOL — SPL является обычным языком программирования высокого уровня общего назначения. Поскольку машинный язык центрального процессо-

ра представляет собой, по существу, реализацию «одни — к — одному» обратной польской записи операторов языка SPL, то имеет смысл вначале познакомиться с языком SPL, а затем перейти к рассмотрению архитектуры центрального процессора.

Язык SPL можно рассматривать как совокупность положений, лежащих в основе таких языков, как APL, ПЛ/1 и ЛИСП. Подобно языку APL, данный язык не содержит типов переменных [отсутствуют операторы описания переменных, а атрибуты (определители) переменных могут быть изменены во время выполнения программы]; в языке SPL имеется только несколько базовых типов данных и структур. Язык SPL подобен языку ПЛ/1 по синтаксису, наличию блочных структур и использованию оператора ON, а языку ЛИСП тем, что первичной структурой данных является список.

Язык SPL оперирует данными двух категорий: скалярами и списками (называемыми структурами). *Скаляр* — это строка символов произвольной длины. Для манипулирования скалярами в языке предусмотрено 20 операций. В качестве иллюстрации приведем следующий фрагмент программы:

```
A ← 22.3;
B ← |abcD$|;
C ← A join B;
OUTPUT C;
```

В результате выполнения этих операций на терминал выводится строка 22.3abcD\$. Знак «|» (вертикальная черта) используется как символ — разделитель литеральных скалярных величин, однако его можно использовать и для скаляров, имеющих синтаксическое строение числовых величин. В языке SPL предусмотрен набор операций и над такими числовыми скалярами. Речь идет о скалярах, подобных следующим:

```
0.5 3.14159EM
486 32.2EX
```

Хотя в общем случае длина скаляра не ограничена (на практике пределом является размер предоставляемой области памяти), максимальная длина используемых числовых скаляров не должна превышать 99 цифр. Числовые величины всегда записываются в памяти машины в виде мантиссы и порядка. Мантисса может содержать 1—99 цифр, а порядок — 1 или 2 цифры. Все арифметические операции выполняются в десятичной системе счисления.

Запись в память числовых величин с переменным (не фиксированным) количеством цифр может привести к увеличению времени обработки и излишнему расходу памяти, поскольку, например, в результате выполнения оператора $A \leftarrow 1/3$; в па-

мать записывается число .3333..., состоящее из 99 цифр. Для устранения данного недостатка в структуру языка SPL дополнительно введены псевдопеременная LIMIT и атрибуты данных EX и EM¹⁾. Псевдопеременная LIMIT может использоваться программистом для задания размерности внутреннего представления результата выполнения последующей операции над числовыми данными. Например, фрагмент программы

```
LIMIT←5;
A←1/3;
```

присваивает переменной A значение .33333EM. Атрибут данных EM, записываемый вместе со значением переменной A, указывает на то, что A представлена не точным значением. Правила выполнения операций над числовыми скалярами определяют длину результата следующим образом:

1) длина результата меньше текущего значения псевдопеременной LIMIT;

2) если хотя бы один операнд представлен неточно (т. е. имеет определитель EM), то длина результата меньше длины самого короткого операнда. Если, согласно данному правилу, отбрасываются ненулевые цифры результата или если один из операндов представлен приближенно, то результат помечается как приближенно определенная величина. Проиллюстрируем это следующим фрагментом программы:

```
LIMIT←5;
A←1/3;      результат=.33333EM
B←1,0;      результат=1.0000EX
LIMIT←10;
C←B/A;      результат=3.0000EM
D←B;        результат=1.000000000EX
E←D*A;      результат=.33333EM
```

Чтобы проверить, представлен ли результат операции с точностью, определяемой псевдопеременной LIMIT, достаточно предусмотреть в программе анализ значения логической переменной LIMIT.

В языке SPL имеется набор логических операций, определенных для скаляров, содержащих только нули или единицы.

СТРУКТУРЫ

Структура — это упорядоченный список, каждый элемент которого может быть скаляром или структурой. Структура может использоваться в качестве массива, но она является более эф-

¹⁾ Аббревиатура EX происходит от английского слова exact — «точный», EM — от английского empirical — «приближенный». — *Прим. перев.*

фективной формой представления данных, чем массив, поскольку ее размер и конфигурация могут изменяться динамически, а ее элементы не обязательно должны иметь одинаковые атрибуты.

Знаки « \langle » и « \rangle » используются в качестве символов-разделителей для определения начала и конца структуры как некоторого литерала, а знак « $|$ » — в качестве символа-разделителя элементов структуры. Следующий фрагмент дает представление о некоторых свойствах структур:

```
A ←  $\langle$ 100|200|300 $\rangle$ ;
OUTPUT A[1];           результат = 100
A[4] ← |WXYZ|;
OUTPUT A;              результат =  $\langle$ 100|200|300|WXYZ $\rangle$ 
A[1] ←  $\langle$ 101|102 $\rangle$ ;
OUTPUT A[1,2];         результат = 102
OUTPUT A;              результат =  $\langle$  $\langle$ 101|102 $\rangle$ |200|300|WXYZ $\rangle$ 
B ← A;
OUTPUT B;              результат =  $\langle$  $\langle$ 101|102 $\rangle$  200|300|WXYZ $\rangle$ 
B[1] ← 100;
OUTPUT B;              результат =  $\langle$ 100|200|300|WXYZ $\rangle$ 
```

Любая ссылка на несуществующий элемент структуры приводит к появлению такого элемента. Например, в результате выполнения оператора $A[6] \leftarrow B$; структура B помещается в шестой элемент структуры A, а элементу A[5] будет присвоено нулевое значение.

СТРУКТУРА ПРОГРАММЫ И ОПЕРАТОРЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

Язык SPL содержит операторы IFTHENELSE и GO TO. Предполагалось включение в язык и оператора организации итеративного цикла LOOP, однако он не был реализован.

В языке SPL имеются процедуры и процедуры-функции. Передача фактических параметров процедуры или процедуры-функции осуществляется указанием имени передаваемого параметра, как это показано в следующем фрагменте программы:

```
A ← 1;
B ← 2;
CALL W(A,A+B);
PROCEDURE W(X, Y);
OUTPUT Y;              результат = 3
X ← X+1;
OUTPUT Y;              результат = 4
END
```

Подобно языкам АЛГОЛ и ПЛ/1, язык SPL ориентирован на блочную структуру программ с использованием правил, определяющих области действия переменных во вложенных процедурах. Однако в отличие от упомянутых языков в языке SPL по умолчанию идентификатор воспринимается как локальный по отношению к процедуре, в которой он используется. Идентификатор может быть определен как синоним по отношению к идентичному идентификатору в вызывающей процедуре с помощью оператора GLOBAL. Оператор GLOBAL X; объявляет идентификатор X в данной процедуре синонимом идентификатора X вызывающей процедуры.

В языке SPL предусмотрено также использование блоков ON, подобных процедурам, за исключением того, что способ обращения к блокам иной. Если в операторе ON указаны одно или несколько имен переменных, то передача управления в соответствующий блок ON производится сразу же по присвоении значения одной из переменных. Если в операторе ON заданы одна или несколько меток операторов, то блок ON вызывается перед передачей управления (с помощью оператора GO TO) на одну из этих меток. Если в операторе ON перечислены одно или несколько имен процедур, то блок ON будет выполнен непосредственно перед вызовом какой-либо из этих процедур. Оператор ON может также использоваться для описания действий при возникновении ситуации INTERRUPT. В этом случае вызов блока ON будет производиться при поступлении прерывания с терминала.

В табл. 8.1 приведен список операторов и операций, используемых в языке SPL.

В качестве иллюстрации структуры программы на языке SPL ниже приводится программа, написанная на учебном языке ПЛ (эта программа заимствована из гл. 6), и эквивалентная программа на языке SPL.

XYZ: PROCEDURE;	B ← 4;
DECLARE A(*) FLOAT;	I ← 1;
DECLARE (B,C) FIXED;	L: IF I LTE B THEN
B = 4;	A[I] ← 1;
ALLOCATE A(1:B);	I ← 1 + I;
A = 1;	GO TO L;
C = B + B * B;	END
CALL ZZZ (B);	C ← B + B * B;
A(B) = C;	CALL ZZZ (B);
ZZZ: PROCEDURE (M);	A[B] ← C;
DECLARE M FIXED;	PROCEDURE ZZZ (M);
C = M + M;	GLOBAL C;
END;	C ← M + M;
END;	END

Таблица 8.1. Список операторов и операций языка SPL

Операторы		Операции	
INPUT OUTPUT GO TO IF Оператор присваивания	SWITCH ON DISABLE ENABLE	+ — • /	BEFORE SAME AFTER JOIN
CALL PAUSE CONTINUE LINK TRAP ¹⁾ Оператор установки начального значения	PROCEDURE BLOCK RETURN GLOBAL SYSTEM ¹⁾ NOTE Обращение к памяти ¹⁾	ABS GREATER GTE EQUALS NEQ LTE LESS	AND OR NOT FORMAT MASK IN

¹⁾ Операторы, имеющие ограниченное применение (только в привилегированных программах).

Цикл в программе на языке SPL, организованной с помощью пяти операторов, может быть опущен и заменен оператором $A \leftarrow \langle 1|1|1|1 \rangle$;

УПРАЖНЕНИЯ

8.1. Является ли аппаратная реализация всех функций системы SYMBOL ее недостатком?

8.2. Что означает выражение $X[2, 3, 4]$, написанное на языке SPL?

8.3. Чем является результат выполнения оператора присваивания $A \leftarrow X[2, 3, 4]$: скаляром или структурой?

8.4. Фрагмент программы на языке SPL содержит группу операторов, оканчивающуюся оператором $A[6] \leftarrow B$. Изобразите схему структуры A после выполнения этого оператора.

ЛИТЕРАТУРА

1. Chesley G. D., Smith W. R., The Hardware-Implemented High-Level Machine Language for SYMBOL, Proceedings of the 1971 Spring Joint Computer Conference Montvale, NJ, AFIPS, 1971, pp. 563—573.
2. Rice R., Smith W. R., SYMBOL — A Major Departure from Classic Software Dominated von Neumann Computing Systems, Proceedings of the 1971 Spring Joint Computer Conference Montvale, NJ, AFIPS, 1971, p. 575—587.
3. Cowart B. E., Rice R., Lundstrom S. F., The Physical Attributes and Testing Aspects of the SYMBOL System, Proceedings of the 1971 Spring Joint Computer Conference Montvale, NJ, AFIPS, 1971, pp. 589—600.

4. Smith W. R. et al., SYMBOL — A Large Experimental System Exploring Major Hardware Replacement of Software, Proceedings of the 1971 Spring Joint Computer Conference Montvale, NJ, AFIPS, 1971, pp. 601—616.
5. Calhoun M. A., SYMBOL Hardware Debugging Facilities, Proceedings of the 1972 Spring Joint Computer Conference Montvale, NJ, AFIPS, 1972, pp. 359—368.
6. Rice R., The Hardware Implementation of SYMBOL, Digest of the Sixth Annual IEEE Computer Society International Conference New York, IEEE, 1972, pp. 27—29.
7. Smith W. R., System Supervision Algorithms for the SYMBOL Computer, Digest of the Sixth Annual IEEE Computer Society International Conference New York, IEEE, 1972, pp. 21—25.
8. Zingg R. J., Richards H., Jr., Operational Experience with SYMBOL, Digest of the Sixth Annual IEEE Computer Society International Conference New York, IEEE, 1972, pp. 31—32.
9. Laliotis T. A., Implementation Aspects of the SYMBOL Hardware Compiler, Proceedings of the First Annual Symposium on Computer Architecture, New York, IEEE, 1973, pp. 111—115.
10. Richards H., Jr., Zingg R. J., The Logical Structure of the Memory Resource in the SYMBOL-2R Computer, Proceedings of the ACM-IEEE Symposium on High-Level Language Computer Architecture, New York, ACM, 1973, pp. 1—10.
11. Anderberg J. W., Smith C. L., High-Level Language Translation in SYMBOL 2R, Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, New York, ACM, 1973, pp. 11—19.
12. Hutchison P. C., Ethington K., Program Execution in the SYMBOL 2R Computer, Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, New York, ACM, 1973, pp. 20—26.
13. Richards H., Jr., Wright C., Jr., Introduction to the SYMBOL 2R Programming Language, Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, New York, ACM, 1973, pp. 27—33.
14. Richards H., Jr., SYMBOL II-R Programming Reference Manual, ISU-CCL-7301, Cyclone Computer Laboratory, Iowa State University, Ames, 1973.
15. Bradley A. C., An Algorithmic Description of the SYMBOL Arithmetic Processor, ISU-CCL-7305, Cyclone Computer Laboratory, Iowa State University, Ames, 1973.
16. Zingg R. J., Richards H., Jr., SYMBOL: A System Tailored to the Structure of Data, ISU-CCL-7302, Cyclone Computer Laboratory, Iowa State University, Ames, 1973.
17. Jones W. E., The Role of the Interface Processor in the SYMBOL IIR Computer System, NSF-OCA-GJ33097-CL7304, Cyclone Computer Laboratory, Iowa State University, Ames, 1973.
18. Dakins M. C., Nonnumeric Processing in the SYMBOL-2R Computer System, NSF-OCA-GJ33097-CL7402, Cyclone Computer Laboratory, Iowa State University, Ames, 1974.
19. Anderberg J. W., Source Program Analysis and Object String Generation Algorithms, and Their Implementation in the SYMBOL-2R Translator, NSF-OCA-GJ33097-CL7410, Cyclone Computer Laboratory, Iowa State University, Ames, 1974.
20. Laliotis T. A., Architecture of the SYMBOL Computer System, In Y. Chu, Ed., High-Level Language Computer Architecture, New York, Academic, 1975, pp. 110—185.
21. Richards H., Jr., Oldehoeft A. E., Hardware-Software Interactions in SYMBOL-2R's Operating System, Proceedings of the Second Annual Symposium on Computer Architecture, New York, 1975, pp. 113—118.
22. Cmelik R. F., Ditzel D. R., The High Level Language Instruction Set of the SYMBOL Computer System, Proceedings of the International Workshop on

- High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 238—246.
23. Ditzel D. R., High Level Language Debugging Tools on the SYMBOL Computer Architecture, University of Maryland, 1980, pp. 247—255.
 24. Ditzel D. R., Program Measurements on a High Level Language Computer, *Computer*, 13(8), 62—72 (1980).
 25. Kwinn W. A., Memory Management Policies for a Hardware Implemented Computer Operating System, MCS72-03642-CL7801, Cyclone Computer Laboratory, Iowa State University, Ames, 1978.
 26. Ditzel D. R., Interactive Debugging Tools for a Block-Structured Programming Language, MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, Ames, 1978.
 27. Rice R., The Chief, Architect's Reflection on Symbol IIR, *Computer*, 14(7), 49—54 (1981).
 28. Ditzel D. R., Reflections on the High-Level Language Symbol Computer System, *Computer*, 14(7), 55—66 (1981).

ГЛАВА 9

АРХИТЕКТУРА ЦЕНТРАЛЬНОГО ПРОЦЕССОРА СИСТЕМЫ SYMBOL

Согласно даинной в гл. 3 классификации вычислительных машин по типу архитектуры, система SYMBOL может быть отнесена к машинам языков высокого уровня с архитектурой типа Б. Набор команд центрального процессора построен по принципу отображения «один к одному» операторов языка SPL, представленных в форме обратной польской записи. Пользуясь терминологией, определенной в гл. 4, можно сказать, что архитектура центрального процессора организована с использованием самоопределяемых (идентифицирующих свой тип) данных; дескрипторов объектов данных; средств управления подпрограммами и динамического управления памятью; данных, занимающих поля переменной длины, а также десятичной арифметики.

О том, что архитектура системы SYMBOL действительно является архитектурой машины языка высокого уровня, свидетельствует тот факт, что в документации описанию набора команд центрального процессора уделяется меньше всего внимания. Отсутствует какой-либо документ, содержащий набор команд. Материал, приведенный в данной главе, составлен по результатам анализа нескольких объектных программ и дискуссий с одним из разработчиков системы. Полный набор команд здесь не обсуждается. Команды этого набора описываются при рассмотрении примера, причем только в той степени, в какой это необходимо для объяснения примера.

ПРЕДСТАВЛЕНИЕ ДАННЫХ

Центральный процессор располагает двумя формами представления скаляров и структур: внешней и внутренней. Внешняя форма представления скаляров и структур в памяти используется при их передаче между центральным процессором и другими процессорами, например транслятором и процессором интерфейса ввода-вывода. Для повышения эффективности обработки данные могут находиться в памяти и во внутренней форме. Поскольку вне центрального процессора единственным представлением данных является внешняя форма (т. е. внутренняя

форма представления данных не находит непосредственного отображения в архитектуре центрального процессора), внутренняя форма представления данных в данной главе не рассматривается; этот вопрос обсуждается в гл. 10.

Внешней формой скаляра является строка, внутренней формой — строка и числовая величина. *Строка* — это линейный список переменных длины из 8-битовых символов, записываемый в целом числе 64-битовых слов. Содержимое строки размещается между служебными символами начала (F5) и конца (F6) строки. Символы, составляющие содержимое строки, представляются в коде ASCII. Старший бит последнего байта в последнем слове строки должен быть единицей; это достигается помещением в этот байт кода F6. Например, скаляр DOG представляется в виде следующей строки:

```
F5 44 4F 47 F6 XX XX F6
```

где XX — условное обозначение неиспользуемого байта слова строки.

Скаляр —289.143ЕМ представляется строкой, занимающей два слова:

```
F5 2D 32 38 39 2E 31 34  
33 45 4D F6 XX XX XX F6
```

Внешняя форма структуры называется ее *линейным представлением*, а внутренняя — *нормальным представлением*. При использовании линейного представления структура в целом ограничена двумя словами, начинающимися со служебных символов начала (FD) и конца (FF) группы. Каждая подгруппа структуры (элемент структуры, более похожий на структуру, чем на скаляр) ограничена двумя словами, начинающимися со служебных символов начала (FC) и конца (FE) подгруппы. Эти символы соответствуют меткам «<» и «>», обозначающим группу в языке SPL. Каждый скалярный элемент структуры записывается в виде строки. Например, структура

```
<101|<201|202>|ABCD>
```

имеет следующее линейное представление:

```
FD XX XX XX XX XX XX XX  
F5 31 30 31 F6 XX XX F6  
FC XX XX XX XX XX XX XX  
F5 32 30 31 F6 XX XX F6  
F5 32 30 32 F6 XX XX F6  
FE XX XX XX XX XX XX XX  
F5 41 42 43 44 F6 XX F6  
FF XX XX XX XX XX XX XX
```

Теперь становятся очевидными причины включения в архитектуру системы отдельного внутреннего представления данных. Например, хотя приведенное выше представление структуры является идеальным для процессора интерфейса ввода-вывода (обеспечивается формат структуры, удобной для отображения на терминале), оно плохо подходит для выполнения операций с индексами элементов структуры, поскольку при этом необходим последовательный просмотр всех слов структуры в процессе поиска требуемого элемента. Следовательно, целью использования отдельного внутреннего представления данных является достижение высокой эффективности их обработки центральным процессором.

Размещение среди данных ограничителей строк, групп и подгрупп структур в первом приближении эквивалентно рассмотренному в гл. 4 использованию тегов, определяющих типы данных.

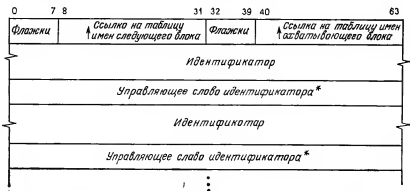
В данном случае данные используются как составная часть их «самоопределителя» (дескриптора), потому что значение данных определяет их тип, а следовательно, и операции, которые могут выполняться над этими данными. Например, если строка состоит только из нулей и единиц, то допускаются логические, арифметические операции и операции над строками символов. Если содержимое строки соответствует синтаксису чисел, то могут выполняться арифметические операции и операции, определенные над строками символов. Если же подобных соответствий нет (строка не является ни последовательностью нулей и единиц, ни числовой последовательностью), то выполнение логических и арифметических операций над строкой не допускает сама машина (аппаратный запрет на выполнение) и выдается сообщение об ошибке в программе.

ТАБЛИЦА ИМЕН

Программа на языке SPL передается центральному процессору для выполнения в виде строки объектного кода и одной или нескольких *таблиц имен*. Используемый здесь термин «строка» не имеет ничего общего с его трактовкой в предыдущем разделе. Транслятор создает одну таблицу имен для каждого блока исходной программы; по назначению эта таблица подобна таблице символических имен и таблице областей переменных в учебной ПЛ-машине, рассмотренной в части II книги.

Таблица имен состоит из *управляющего слова таблицы* (block control word), за которым следуют элементы для каждого идентификатора блока программы (например, метки оператора, имени переменной, имени процедуры). Формат таблицы изображен на рис. 9.1, а описание назначения отдельных би-

тов ее первого слова (управляющего слова таблицы имен) приведено в табл. 9.1. Назначение большинства полей этого слова понятно без дополнительных пояснений. Единичное значение бита 4 указывает на то, что в данном блоке программы разре-



* Или текущее значение величины, определяемой идентификатором, если это скаляр, содержащий не более шести символов.

Рис. 9.1. Формат таблицы имен, используемой в системе SYMBOL.

шается выполнять команды, допустимые только для системных программ.

Таблица 9.1. Управляющее слово таблицы имен

Номера битов	Назначение
0	Всегда 1 (определяет управляющее слово)
1	Всегда 1 (определяет управляющее слово таблицы имен)
2	Всегда 0
3	Не используется
4	1, если процедура привилегированная
5	Всегда 1 (используется только на этапе трансляции)
6	В управляющем слове содержится ссылка (биты 8—31) на таблицу имен следующего блока программы — вызываемой процедуры
7	В управляющем слове содержится ссылка (биты 40—63) на таблицу имен предшествующего блока программы — вызывающей процедуры
8—31	Адрес таблицы имен следующего блока программы
32—36	Не используются
37	1, если блок активирован
38	1 при рекурсивном вызове блока
39	Не используется
40—63	Адрес таблицы имен вызывающего блока программы

Таблица 9.2. Управляющее слово идентификатора

Номера битов	Назначение битов
0	Всегда 1 (определяет управляющее слово)
1	Обычно 0 (определяет управляющее слово идентификатора); 1 в сочетании с единичными значениями битов 1—3 ¹⁾
2	Указатель управляющего слова последнего идентификатора в таблице имен ¹⁾
3—4	00, если имеет место обычная локальная переменная 01, если имеет место глобальная переменная 10, если имеет место инициализируемая локальная переменная или метка, процедура либо параметр 11 — системой не воспринимается
5	1, если величина, определяемая идентификатором, находится в строке объектного кода
6	1, если имеет место структура или элемент структуры
7	1, если биты 32—37 содержат дополнительные флажки
8—31	См. табл. 9.3
32	1, если разрешено использование блока ON ²⁾
33	Не используется ²⁾
34	1, если идентификатор определяет метку ²⁾
35	1, если идентификатор определяет процедуру ²⁾
36	1, если идентификатор определяет параметр ²⁾
37	1, если имеется ссылка на идентификатор в операторе ON ²⁾
38—39	Не используются ²⁾
40—63	См. табл. 9.3

¹⁾ Если биты 0—3 равны 1111, то эти биты и остальная часть управляющего слова идентификатора интерпретируются по-другому; они указывают, что величина, определяемая идентификатором, находится в битах 0—63 этого слова.

²⁾ Если бит 7 равен 0, то в битах 32—39 размещается последний использованный индекс элемента структуры (см. гл. 10).

Элементы таблицы имен состоят из двух или более слов. Первой частью элемента является само имя идентификатора, представленное в виде строки скаляров и занимающее одно или несколько слов. Имя используется при выдаче пользователю сообщения об ошибке и при выполнении оператора OUTPUT DATA (отображающего на дисплее как имя идентификатора, так и значение определяемой им величины). Вторая часть элемента таблицы представляет собой *управляющее слово идентификатора*, которое по назначению подобно дескрипторам, описываемым в гл. 4. В упрощенном виде идея использования этих управляющих слов состоит в том, что адреса операндов в объектной программе указывают управляющие слова идентификаторов, а последние указывают значения самих операндов. Назначение битов управляющего слова идентификатора поясняется в табл. 9.2 и 9.3.

Существуют два основных формата управляющего слова идентификатора. Если первые 4 бит имеют единичные значения,

Таблица 9.3. Адресные поля управляющего слова идентификатора

Тип идентификатора	Содержимое адресных полей
Локальная переменная (биты 3—4 равны 00 или 10)	Биты 8—31: начальный адрес местоположения переменной (если она задана) Биты 40—63: адрес блока ON, связанного с этой переменной (если этот блок имеется)
Локальная переменная типа структуры, не связанная с блоком ON (биты 3—4 равны 00 или 10, бит 6 равен 1, бит 7 равен 0)	Биты 8—31: начальный адрес местоположения переменной (если она задана) Биты 32—39: значение индекса последнего использованного элемента структуры Биты 40—63: начальный адрес последнего использованного элемента структуры
Глобальная переменная (биты 3—4 равны 01)	Биты 8—31: адрес управляющего слова идентификатора соответствующей переменной в вызывающем блоке программы Биты 40—63: не используются
Параметр (биты 7 и 36 равны 1)	Биты 8—31: адрес управляющего слова идентификатора или объектного кода соответствующего фактического параметра (если он задан) Биты 40—63: не используются
Метка (биты 7 и 34 равны 1)	Биты 8—31: адрес объектного кода Биты 40—63: адрес связанного с данной меткой блока ON (если этот блок имеется)
Процедура (биты 7 и 35 равны 1)	Биты 8—31: адрес объектного кода Биты 40—63: адрес связанного с данной процедурой блока ON (если этот блок имеется)

то величина, определяемая идентификатором, записана непосредственно в управляющем слове. В противном случае управляющее слово идентификатора имеет формат, описанный в табл. 9.2, и если в данный момент идентификатору присвоено значение, то биты 8—31 управляющего слова — это виртуальный адрес первого слова поля, в котором находится это значение. Если выполняется машинная команда, присваивающая идентификатору определенное значение, то центральный процессор сначала анализирует управляющее слово идентификатора и присваиваемую ему величину. Если значение бита 7 равно нулю (идентификатор представляет собой переменную, не связанную с блоком ON), а назначаемая величина — скаляр из шести символов или меньше, то эта величина записывается в управляющее слово идентификатора. В противном случае с помощью контроллера памяти для величины, присваиваемой идентификатору, выделяется область памяти, адрес которой записывается в биты 8—31. Исключением являются случаи, когда идентификатор определяет глобальную переменную или фор-

мальный параметр процедуры и как следствие этого в управляющее слово идентификатора помещается адрес фактической переменной, соответствующей данной глобальной переменной или формальному параметру. Например, если локальной переменной присвоено значение 123, то управляющее слово идентификатора имеет следующий вид:

F5 31 32 33 F6 XX XX F6

Если же локальной переменной присваивается величина —123.56789, то управляющее слово идентификатора может выглядеть таким образом:

80 00 77 78 XX XX XX XX

Предполагается, что область с адресом 7778 выделена для размещения значения переменных.

Для иллюстрации процесса формирования транслятором таблиц имен и строки объектного кода воспользуемся приводимой ниже программой

```

B ← 4;
I ← 1;
L: IF I LTE B THEN
  A[ I ] ← 1;
  I ← I + 1;
  GO TO L;
END
C ← B + B * B;
CALL ZZZ(B);
A[B] ← C;
PROCEDURE ZZZ (M);
GLOBAL C;
C ← M + M;
END

```

На рис. 9.2 показаны таблицы имен, полученные в результате трансляции программы. Числа, расположенные слева от каждого элемента таблиц, определяют его виртуальный адрес. Хотя таблица имен воспринимается транслятором и центральным процессором как список, записанный в смежных областях памяти, элементы таблицы располагаются в виртуальной памяти не обязательно последовательно. Этот факт не очевиден только для контроллера памяти из-за метода, посредством которого контроллер отображает списковое представление структуры реальной памяти в последовательную виртуальную память. (Этот метод рассматривается в гл. 10.)

Поскольку атрибуты и размеры переменных могут изменяться динамически, транслятор не резервирует память для их раз-

мещения. Следовательно, биты 8—31 управляющего слова идентификатора для всех локальных переменных и параметров первоначально имеют нулевые значения. Идентификаторы L и ZZZ относятся к метке и процедуре соответственно; таким об-

Виртуальный адрес		Идентифи- катор
2220	C6 00 22 88 00 00 00 00	
2221	42 XX XX XX XX XX XX XX	B
2222	80 00 00 00 XX XX XX XX	
2223	49 XX XX XX XX XX XX XX	I
2224	80 00 00 00 XX XX XX XX	
2225	4C XX XX XX XX XX XX XX	L
2226	91 00 30 28 20 XX XX XX	
2227	41 XX XX XX XX XX XX XX	A
2270	80 00 00 00 XX XX XX XX	
2271	43 XX XX XX XX XX XX XX	C
2272	80 00 00 00 XX XX XX XX	
2273	5A 5A 5A XX XX XX XX XX	ZZZ
2274	B1 00 30 72 10 XX XX XX	
2288	C5 00 00 00 00 00 22 20	
2289	4D XX XX XX XX XX XX XX	M
228A	91 00 00 00 08 XX XX XX	
228B	43 XX XX XX XX XX XX XX	C
228C	A3 00 22 72 XX XX XX XX	

Рис. 9.2. Таблицы имен для рассматриваемой программы.

разом их управляющее слово в таблице имен указывает на определенную точку в строке объектного кода. (Объектный код для данной программы приводится в следующем разделе. Объектный код для метки L начинается с адреса 3028, а для процедуры ZZZ — с адреса 3072.) Заметим также, что переменная C описана как глобальная во второй таблице имен; следовательно, управляющее слово идентификатора для переменной C

в этой таблице адресует управляющее слово идентификатора **C** в таблице имен внешней процедуры.

ОБЪЕКТНЫЙ КОД ПРОГРАММЫ

Набор команд центрального процессора системы **SYMBOL** является стеко-ориентированным и подобен набору команд учебной ПЛ-машины. Принципиальные различия между наборами команд этих двух систем заключаются, во-первых, в способе адресации операндов: вместо использования адресации на лексическом уровне команды системы **SYMBOL** адресуют свои операнды с помощью управляющего слова идентификатора; во-вторых, набор команд системы **SYMBOL** менее эффективно использует память, и, в-третьих, в системе **SYMBOL** отсутствуют операции над массивами (за исключением операции присвоения значения).

Каждая команда **SYMBOL** начинается с 8-битового кода операции. В зависимости от кода операции оставшаяся часть команды имеет один из следующих четырех форматов:

1) операнд отсутствует, длина команды составляет 32 бит; последние 24 бит не используются;

2) операнд представляет собой данные; длина команды равна 32 бит; последние 24 бит — виртуальный адрес управляющего слова идентификатора;

3) операнд представляет собой команду; длина команды равна 32 бит; последние 24 бит — виртуальный адрес команды;

4) операнд представляет собой литерал; длина команды — переменная, равная целому числу слов; код операции располагается на границе слова; код операции — служебный символ начала строки, начала структуры или начала подгруппы структуры; операнд — оставшаяся часть строки или линейного представления структуры.

Обычно команды размещаются по две в одном слове памяти; при этом код операции команды должен находиться на границе слова или полуслова. Команды, которые расположены не на границе слова, — неадресуемы. Для того чтобы к подобной команде могла адресоваться другая команда, первую необходимо выровнять на границу слова. Это осуществляется включением в соответствующее место программы команд типа **NOP** (**NO OPERATION** — **НЕТ ОПЕРАЦИИ**). Этот же прием применяется для компенсации дефектов конструкции транслятора, обуславливающих расположение команд одного типа на границе слова, а другого типа — вне таких границ.

В качестве примера рассмотрим объектный код приведенной выше программы на языке **SPL** вместе с таблицами имен, представленными на рис. 9.2. Подобно команде **LINE** учебной

ПЛ-машины, объектный код системы SYMBOL указывает соответствующие номера операторов исходной программы, которые могут быть использованы для отладки последней. В данном случае это достигается хранением исходной программы в памяти во время выполнения ее объектного кода. На рис. 9.3 показано, как представляется исходная программа в памяти машины.

Виртуальный адрес	Содержимое слова	Виртуальный адрес	Содержимое слова
1020	B ← 4; I ←	1028	CALL ZZZ
1021	I; L; 1F	1029	(B); A B
1022	I L T E B	102A	+C; PRO
1023	THEN A I	102B	EDURE Z
1024	← I; I ← I	102C	ZZ(M); G
1025	+I; GO T	102D	LOBAI C;
1026	O I; END	102E	C ← M + M; E
1027	C ← B + B * B;	102F	ND

Рис. 9.3. Представление исходной программы в памяти.

3020	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
	90	002220	Команда BLOCK
3021	D0	002222	Команда NTP для переменной B
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
3022	F5	34F6XX	Строка данных (число 4)
	XX	XXXXXF6	
3023	DF	XXXXXX	Команда присвоения значения
	BB	XXXXXX	Команда конца оператора исходной программы
3024	D9	001020	Команда указания оператора исходной программы
	D0	002224	Команда NTP для переменной I
3025	F5	31F6XX	Строка данных (число 1)
	XX	XXXXXF6	
3026	DF	XXXXXX	Команда присвоения значения
	BB	XXXXXX	Команда конца оператора исходной программы
3027	D9	001021	Команда указания оператора исходной программы
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
3028	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
	90	002220	
3029	D0	002224	Команда NTP для переменной I
	D0	002222	Команда NTP для переменной B
302A	9A	XXXXXX	Команда сравнения на «меньше» или «равно»
	B5	003067	Переход, если ЛОЖНО
302B	D0	002270	Команда NTP для идентификатора структуры A
	D0	002222	Команда NTP для переменной I
302C	DD	XXXXXX	Команда поиска элемента структуры
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
302D	F5	31F6XX	Строка данных (число 1)
	XX	XXXXXF6	
302E	DF	XXXXXX	Команда присвоения значения
	BB	XXXXXX	Команда конца оператора исходной программы

Рис. 9.4. Первая часть объектного кода программы.

Первая часть объектной программы приведена на рис. 9.4. В первой колонке слева указан виртуальный адрес каждого слова. Вторая и третья колонки содержат код операции и адреса операндов. Как и ранее, X обозначает неиспользуемые байты команды. Четвертая колонка содержит комментарий к каждой команде. Последовательность машинных команд не обязательно записывается в смежных ячейках виртуальной памяти. Это является очевидным только для контроллера памяти; центрально-

му процессору команды представляются расположенными в памяти последовательно в виде линейного списка.

Первой выполняемой командой программы является команда **BLOCK**. Транслятор всегда генерирует команду **BLOCK** во второй части слова, располагая перед ней команду **НЕТ ОПЕРАЦИИ**. Команда **BLOCK** указывает адрес (002220) управляющего слова таблицы имен.

Следующие пять команд соответствуют оператору исходной программы **B←4**. Первая из них — команда **NTP**. Идентификаторы, появляющиеся в тексте исходной программы, транслируются в команду **NTP**, указывающую положение управляющего слова данного идентификатора в таблице имен. Интерпретация команды зависит от атрибутов идентификатора, находящихся в таблице имен. Например, если идентификатор ссылается на переменную, то **NTP** — команда загрузки данных в стек; если на процедуру, то **NTP** — команда вызова процедуры. В данном случае по команде **NTP** в стек центрального процессора загружается слово *связи*. Слово *связи* содержит адрес (002222) управляющего слова идентификатора **B**, а также байт, идентифицирующий этот элемент стека (в данном случае величину **E0**, означающую «слово *связи* для простой переменной»).

Заметим, что следующая выполняемая команда имеет формат строки данных, описанный в начале главы. (Команде предшествует команда **NOP**, обеспечивающая выравнивание строки данных на границу слова.) Любая строка данных, встречающаяся в потоке машинных команд, загружается в стек.

Команда **ASSIGN** (адрес 3023) выделяет память для величины, находящейся в данный момент на вершине стека, модифицирует управляющее слово идентификатора, адресуемое следующим элементом стека (так что оно теперь указывает выделенную область памяти), и удаляет элемент, находящийся на вершине стека. В данном случае память выделена не будет, поскольку величина переменной (число 4) достаточно мала и может быть записана непосредственно в управляющее слово идентификатора по адресу 002222. Команда **END-OF-STATEMENT** производит очистку стека и разрешает вызов любого блока **ON**, ожидающего в данный момент своего выполнения. Команда **SOURCE-POINTER** указывает адрес конца соответствующего оператора исходной программы; она воспринимается как команда **НЕТ ОПЕРАЦИИ**.

Команды, расположенные начиная со второй половины слова с адресом 3024 до слова с адресом 3027, полностью подобны предшествующим и относятся к оператору **I←1**.

Следующая команда **BLOCK** обусловлена наличием в исходной программе метки **L**. Подобная команда необходима для каждой метки исходной программы, поскольку дает машине

возможность «настроиться» на ситуацию, при которой на метку ссылается оператор GO TO, расположенный в другом (внутреннем) блоке. Заметим, что управляющее слово идентификатора L в таблице имен указывает слово с адресом 3028.

Следующие три команды вычисляют логическую величину, соответствующую условию, указанному в операторе IF, и помещают ее на вершину стека. Если эта величина имеет значение «ложно», то управление передается на выполнение объектного кода, задаваемого текстом исходной программы, следующим за ELSE (предложение ELSE отсутствует в языке SPL, поэтому управление передается на выполнение объектного кода следующего оператора).

Команды, расположенные начиная с адреса 302B, соответствуют оператору $A[I] \leftarrow 1$, который с помощью индекса ссылается на элемент структуры. Команда SUBSCRIPT преобразует величину, находящуюся на вершине стека в целое число, и осуществляет поиск в стеке первого слова для структуры, заменяя эту величину и расположенные над ней величины (значения индекса или индексов) на адрес элемента, к которому производится обращение. (Если используется множественное индексирование, то после команд NTP или STRING должна быть сгенерирована команда INTEGERIZE для преобразования всех индексов, кроме последнего, в целые величины.) Поскольку к этому моменту величина $A[I]$ еще не известна, и процессор еще не знает, какой объем памяти следует выделить для данного элемента структуры, рассматриваемая операция — индексирование — задерживается до выполнения команды ASSIGN, расположенной по адресу 302E.

Объектный код оператора $I \leftarrow I + 1$ состоит из восьми команд, расположенных в памяти, начиная со второй половины слова с адресом 302F (рис. 9.5). Команда ADD выполняет проверку формата числовых операндов и инициирует сообщение об ошибке, если хотя бы один операнд является не числовым. И опять-таки вследствие особенностей построения транслятора в объектный код программы включается команда НЕТ ОПЕРАЦИИ. Команда GO TO, расположенная по адресу 3065, соответствует оператору GO TO исходной программы. В этот момент адрес управляющего слова идентификатора для метки L находится в стеке; само управляющее слово содержит адрес 3028.

Команды, расположенные по адресам 3067—306A, соответствуют оператору $C \leftarrow B + B * V$ и выполняют указанные операции непосредственно. Команда НЕТ ОПЕРАЦИИ находится по адресу 3066, поскольку передача управления (в обход отсутствующего предложения ELSE оператора IF) осуществляется команде NTP для переменной C. Все команды, выполняющие арифметические операции, проверяют, корректно ли заданы

302F	D9	001024	Команда указания оператора исходной программы
	D0	002224	Команда NTP для переменной I
3060	D0	002224	Команда NTP для переменной I
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
3061	F6	31F6XX	Строка данных (число 1)
	XX	XXXXF6	
3062	AB	XXXXXX	Команда сложения
	DF	XXXXXX	Команда присвоения значения
3063	BB	XXXXXX	Команда конца оператора исходной программы
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
3064	D9	001025	Команда указания оператора исходной программы
	D0	002226	Команда NTP для метки L
3065	95	XXXXXX	Команда перехода
	BB	XXXXXX	Команда конца оператора исходной программы
3066	D9	001026	Команда указания оператора исходной программы
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
3067	D0	002272	Команда NTP для переменной C
	D0	002222	Команда NTP для переменной V
3068	D0	002222	Команда NTP для переменной V
	D0	002222	Команда NTP для переменной V
3069	AA	XXXXXX	Команда умножения
	AB	XXXXXX	Команда сложения
306A	DF	XXXXXX	Команда присвоения значения
	BB	XXXXXX	Команда конца оператора исходной программы
306B	D9	001027	Команда указания оператора исходной программы
	D0	002273	Команда NTP для идентификатора ZZZ
306C	D4	002222	Команда передачи параметра
	BB	XXXXXX	Команда конца оператора исходной программы

Рис. 9.5. Вторая часть объектного кода программы.

операнды и не превышает ли результат операции предельно допустимого значения. В данном случае подобных ситуаций не возникает.

Последние три команды (рис. 9.5) реализуют оператор CALL. Команда NTP для идентификатора ZZZ интерпретируется как вызов процедуры. При этом процессор осуществляет поиск команд с параметрами процедуры (которые загружаются в стек) и команд пересылки (которые выполняются). При обнаружении команды, соответствующей точке возврата, устанавливается связь с таблицей имен для этой новой процедуры (посредством команды BLOCK, с которой она начинается), команды с параметрами извлекаются из стека и используются для присвоения начальных значений управляющим словам идентификаторов формальных параметров вызываемой процедуры. Вход в блок осуществляется загрузкой в стек информации о состоянии вызывающего модуля, получения памяти для нового стека и установления связи нового стека с предыдущим.

Поскольку передача процедуре фактических параметров осуществляется по имени (а не путем передачи значений этих параметров) и в связи с тем, что фактические параметры могут представлять собой выражения, включающие знаки операций, значения параметров процедуры не могут быть определены в момент ее вызова. Вместо этого в управляющие слова идентификаторов фактических параметров помещается ссылка на одну или более команд объектной программы, которые выполняются при каждом обращении к этим параметрам. Однако

в этом случае фактический параметр является переменной; поэтому управляющее слово идентификатора формального параметра модифицируется таким образом, чтобы указывать на управляющее слово идентификатора фактического параметра. Так, управляющее слово идентификатора, расположенное по адресу 2289 (рис. 9.2), будет указывать управляющее слово идентификатора, находящееся по адресу 002222. Пример, объясняющий механизм передачи процедуре фактических параметров (представляющих собой выражения) по имени, приводится в следующем разделе.

306D	D9	001029	Команда	указания оператора исходной программы
	D0	002270	Команда	NTP для переменной A
306E	D0	002222	Команда	NTP для переменной B
	DD	XXXXXX	Команда	поиска элемента структуры
306F	D0	002272	Команда	NTP для переменной C
	DF	XXXXXX	Команда	присвоения значения
3070	BB	XXXXXX	Команда	конца оператора исходной программы
	00	XXXXXX	Команда	НЕТ ОПЕРАЦИИ
3071	D9	00102A	Команда	указания оператора исходной программы
	D7	003078	Команда	перехода
3072	00	XXXXXX	Команда	НЕТ ОПЕРАЦИИ
	90	002288	Команда	BLOCK
3073	D9	00102C	Команда	указания оператора исходной программы
	D0	00228C	Команда	NTP для переменной C
3074	D0	00228A	Команда	NTP для переменной M
	D0	00228A	Команда	NTP для переменной M
3075	AB	XXXXXX	Команда	сложения
	DF	XXXXXX	Команда	присвоения значения
3076	BB	XXXXXX	Команда	конца оператора исходной программы
	00	XXXXXX	Команда	НЕТ ОПЕРАЦИИ
3077	D9	00102E	Команда	указания оператора исходной программы
	B7	XXXXXX	Команда	BLOCK
3078	B7	XXXXXX	Команда	BLOCK

Рис. 9.6. Третья часть объектного кода программы.

Согласно рис. 9.6, управление теперь передается по адресу 3072 (величина, находящаяся в управляющем слове идентификатора ZZZ). При выполнении команды NTP, расположенной по адресу 3073, машина выясняет, что переменная C — глобальная, и поэтому просмотр цепочки управляющих слов идентификатора, на которую указывает управляющее слово, осуществляется в обратном порядке. Поиск продолжается до тех пор, пока не встретится управляющее слово, не относящееся к глобальной переменной. Как только такое слово найдено, оно загружается в стек. При выполнении команд, содержащихся в слове с адресом 3074, машина выясняет, что определяемые этими командами управляющие слова идентификаторов являются формальными параметрами процедуры, и загружает адрес соответствующего управляющего слова идентификатора фактического параметра в стек. Стек теперь содержит два указателя управляющего слова идентификатора B и указатель управляющего слова идентификатора C — переменная C определена во внешней процедуре.

Команда END-BLOCK, расположенная в слове с адресом

Таблица 9.4. Сопоставление размеров программ

Вычислительная система	Количество выполняемых команд	Размер объектного кода, байт	Общий размер программы, байт
Система SYMBOL	81	324	396
Учебная ПЛ-машина	62	96	369
Система 370	303 ¹⁾	564	5912
Система 370 ²⁾	112 ¹⁾	360	5496

¹⁾ И еще 90 000 команд, выделяющих в начале программы память для организации вызова подпрограмм и возврата из них.

²⁾ Вариант программы на языке ПЛ/1 без операторов ALLOCATE и контроля индексирования массивов.

3077, завершает выполнение процедуры. При этом текущее содержимое стека уничтожается, восстанавливается информация, сохраняемая в предыдущем стеке, и осуществляется возврат управления по адресу 306D. Заметим, что адреса локальных переменных в таблице имен, относящейся к завершенной процедуре, не изменяются, и им не присваиваются нулевые значения: в этом смысле локальные переменные являются статичными.

Команды, находящиеся по адресам 306D—3070, присваивают значение элементу структуры A[B], после чего управление передается в обход тела внутренней процедуры и программа завершается.

В табл. 9.4 приводится сравнение характеристик рассмотренной объектной программы с объектными программами для эквивалентных программ, написанных на учебном языке ПЛ и языке ПЛ/1 Системы 370. Записываемая в память копия исходной программы, составленной на языке SPL, не входит в числовые величины таблицы, относящиеся к системе SYMBOL, для достижения наибольшей точности проводимого сравнения. Причиной более низкой эффективности системы SYMBOL по сравнению с учебной ПЛ-машиной является нерациональное конструирование ее набора команд. Так, 8-битовая команда дополняется 24 неиспользуемыми битами, производится выравнивание генерируемых транслятором команд при размещении их в памяти.

ПЕРЕДАЧА ФАКТИЧЕСКИХ ПАРАМЕТРОВ ВЫЗЫВАЕМОЙ ПРОЦЕДУРЕ

Поскольку, согласно языку SPL, передача фактических параметров вызываемой процедуре осуществляется путем указания их имен, формальным параметрам процедуры не могут быть присвоены исходные значения во время ее вызова. Поэтому при каждом использовании параметра его значение подлежит определению. Так, оператор языка SPL

CALL COST(1+K,20*M)

формирует фрагмент объектной программы, представленный на рис. 9.7, причем символы ZZZZZZ обозначают местоположение ссылок на таблицу, содержащую имена подлежащих определению величин.

1120	D0	ZZZZZZ	Команда NTP для идентификатора COST
	D7	001126	Команда перехода
1121	D0	ZZZZZZ	Команда NTP для переменной I
	D0	ZZZZZZ	Команда NTP для переменной K
1122	AB	XXXXXX	Команда сложения
	D8	XXXXXX	Команда окончания вычисления параметра
1123	F5	3230F6	Строка данных (число 20)
	XX	XXXXF6	
1124	D0	ZZZZZZ	Команда NTP для переменной M
	AA	XXXXXX	Команда умножения
1125	D8	XXXXXX	Команда окончания вычисления параметра
	00	XXXXXX	Команда НЕТ ОПЕРАЦИИ
1126	D5	001123	Команда присвоения параметру значения выражения, его представляющего
	D5	001121	Команда присвоения параметру значения выражения, его представляющего
1127	BB	XXXXXX	Команда конца оператора исходной программы

Рис. 9.7. Объектный код фрагмента программы, передающего параметры по имен.

Команды INDIRECT-PARAMETER обеспечивают присвоение исходных значений формальных параметров в таблице имен для процедуры COST таким образом, чтобы эти параметры указывали не на управляющие слова идентификаторов фактических параметров, а на строки объектного кода выражений, вычисляющих значения фактических параметров. Когда на такой параметр производится ссылка в вызываемой процедуре, соответствующая строка объектного кода выполняется до тех пор, пока не встретится команда PARAMETER-RETURN. Например, всякий раз, когда в процедуре COST для первого параметра необходимо выполнение команды NTP, происходит выполнение команд NTP I, NTP K и ADD.

УПРАЖНЕНИЯ

9.1. Одной из внутренних форм представления скаляров является числовое представление. Хотя этот вопрос еще не обсуждали, сделайте предположение о том, на что должно быть похоже такое представление.

9.2. Каким является линейное представление структуры (00000000|1111111)?

9.3. Опишите содержимое управляющего слова идентификатора, расположенного по адресу 2224, после выполнения команды, находящейся по адресу 3026, для программы, приведенной на рис. 9.4.

9.4. Опишите содержимое управляющего слова идентификатора, расположенного по адресу 2224, если переменной I было присвоено значение 1234.5678.

9.5. Из какого управляющего слова идентификатора будет получена информация о величине M при выполнении объектного кода, соответствующего оператору $C \leftarrow M + M$? Содержимое какого управляющего слова идентификатора изменит этот оператор?

ГЛАВА 10

ВНУТРЕННЯЯ СТРУКТУРА И ВЗАИМОСВЯЗЬ ПРОЦЕССОРОВ СИСТЕМЫ SYMBOL

По содержанию эта глава, посвященная практической реализации процессоров системы SYMBOL и их взаимосвязи в процессе функционирования, является как бы отступлением от основной темы книги — анализа архитектуры ЭВМ. Целесообразность рассмотрения данного предмета обусловлена двумя обстоятельствами. Во-первых, архитектура центрального процессора вычислительной системы в существенной мере определяет реализацию системы в целом, и в частности средства управления памятью, а следовательно, заслуживает первостепенного внимания. Во-вторых, поскольку одним из уникальных свойств системы SYMBOL является распределенная мультипроцессорная обработка особого типа с функциональной ориентацией отдельных процессоров, проведение более полного анализа организации системы является оправданным.

ОСНОВНАЯ ШИНА

Так как основная шина служит средством взаимосвязи всех процессоров системы (рис. 8.1), первым шагом на пути изучения внутреннего функционирования системы SYMBOL должно быть ознакомление с назначением и техникой использования этой шины. По основной шине передаются следующие сигналы:

- 1) запросы на обращение к памяти от процессора к контролеру памяти (цикл запроса на обращение к памяти);
- 2) результаты обращения к памяти из контроллера памяти к процессору (цикл передачи ответа на обращение к памяти);
- 3) сигнал управления от процессора к супервизору (цикл обмена сигналами управления);
- 4) сигнал управления от супервизора к процессору (цикл обмена сигналами управления);
- 5) данные между процессорами, входящими в состав центрального процессора (так называемый неявный цикл).

Основная шина состоит из 111 линий. Назначение линий указано в табл. 10.1. Шина используется в соответствии с системой

Таблица 10.1. Назначение линий основной шины

Номера линий	Обозначение линий	Назначение
1		Не используется
2—7	MC	Функция обращения к контроллеру памяти и коды завершения
8—12	MT	Номер терминала
13—36	MA	Адрес памяти
37—100	MD	Данные
101		Сброс системы
102		Тактовые импульсы
103	MP1	Запрос обращения к памяти супервизора системы
104	MP2	Запрос обращения к памяти процессора интерфейса ввода-вывода
105	MP3	Запрос обращения к памяти центрального процессора
106	MP4	Запрос обращения к памяти транслятора
107	MP5	Указание на то, что следующим будет цикл передачи ответа на обращение к памяти
108	MP6	Запрос цикла обмена сигналами управления
109	MP7	Указание, что контроллер памяти свободен
110	MP8	Запрос обращения к памяти регенератора памяти
111		Не используется

приоритетов; линиями, обладающими определенными приоритетами, являются MP1—MP8. Наивысший приоритет присвоен линии MP1, самый низкий — линии MP8. Когда процессору нужна основная шина, он запрашивает одну из линий MP1—MP8. Например, если центральному процессору необходимо обратиться к контроллеру памяти, он активирует линию MP3, а если необходима связь с супервизором — линию MP6. Если в данный момент ни одна из линий, обладающих приоритетом, не занята, то шина используется процессором в течение следующего цикла — периода между тактовыми импульсами устройства синхронизации. Если же нужная линия, имеющая приоритет, занята, то процессор пропустит один цикл и повторит запрос.

Предположим, что центральному процессору требуется передать слово данных в память. Для этого он возбуждает линию MP3 и проверяет состояние линий MP1, MP2 и MP7. Если линии MP1 и MP2 неактивны, а на линии MP7 сигнал установлен (это указывает на то, что контроллер памяти не занят обработкой другого запроса), то центральный процессор передаст адрес памяти на линии MA, данные — на линии MD, код требуемой операции обращения к памяти на линии MC, идентификатор терминала или пользователя (для которого выполня-

ется обращение к памяти) на линии МТ. Затем контроллер памяти осуществляет сброс сигнала на линии МР7 (линия перестает быть активной) и продолжает обрабатывать запрос. В это время шина может использоваться для передачи данных между процессорами, но не для обращения к памяти. По окончании обработки запроса контроллер памяти устанавливает сигнал на линии МР5. Во время следующего цикла контроллер памяти передает на линии МС код завершения, и, если в ответ на обращение к памяти возвращается адрес соответствующей области (что является обычным для систем со списковой организацией памяти), этот адрес поступает на линии МА. В это время центральный процессор проверяет состояние линии МР5. Обнаружив сигнал на этой линии, процессор получает тем самым информацию о том, что во время следующего цикла в шину будут переданы результаты обработки запроса контроллером памяти.

Если необходимо передать сигнал управления от процессора к супервизору или от супервизора к одному или нескольким процессорам, то активируется линия МР6, и если линии МР1—МР5 неактивны, то управляющая информация передается на линии МД. Такой режим работы линии называется циклом обмена сигналами управления. Во время этого режима линию потенциально могут использовать все процессоры. Такая возможность обеспечивается за счет разделения во время этого цикла отдельных линий из группы линий МД между различными группами устройств. Например, во время цикла обмена сигналами управления по линиям 37—40 передаются сигналы от супервизора к центральному процессору, по линиям 41—44 — от центрального процессора к супервизору, по линиям 45—48 — от супервизора к транслятору, по линиям 49—52 — от транслятора к супервизору и т. д.

Как упоминалось выше, линии МС используются во время цикла запроса на обращение к памяти для передачи функций обращения к контроллеру памяти. Сигналы на линиях 2—5 определяют одну из 14 возможных функций обращения к логической памяти, которая должна быть реализована, а сигналы на линиях 6—7 указывают, к какому списку страниц виртуальной памяти пользователя относится данный запрос (последние из упомянутых сигналов позволяют минимизировать фрагментацию памяти; использование обеих этих групп линий описывается в следующем разделе). Во время цикла передачи ответа на обращение к памяти контроллер памяти передает на линии МС код завершения. Например, наличие кода 000011 свидетельствует об успешном выполнении запроса, кода 000001 — об отсутствии требуемой страницы памяти, код 100000 указывает на сбой оборудования при обращении к памяти.

УПРАВЛЕНИЕ ПАМЯТЬЮ

Наиболее существенной особенностью системы SYMBOL является, по-видимому, использование высокоорганизованной, гибкой и динамичной системы управления памятью. Как отмечено в гл. 8, эта система реализуется главным образом в виде контроллера памяти и в меньшей степени как регенератор памяти и супервизор.

Системе SYMBOL присуща иерархическая организация памяти, верхний уровень которой занимает так называемая *логическая память*. Такая организация позволяет рассматривать реальную память как неограниченное число линейных списков неограниченной длины или строк, состоящих из 64-битовых слов. Следующей в иерархии является виртуальная память, представляющая реальную память как непрерывное адресное пространство (вектор) из 16 777 216 64-битовых слов. Функции контроллера памяти сводятся к отображению логической памяти в виртуальную память. Последняя разделена на страницы, каждая из которых содержит 256 слов (2048 байт или символов). Одна из функций супервизора состоит в отображении страниц виртуальной памяти в память третьего уровня, т. е. в основную память, являющуюся физически реальной.

ЛОГИЧЕСКАЯ ПАМЯТЬ

Логическая память — это такое представление памяти, которое возникает у процессоров системы с помощью контроллера памяти. Лучшим способом описания структуры логической памяти является определение 14 функций обращения к памяти, реализуемых контроллером памяти (табл. 10.2).

Согласно гл. 9, такие элементы объектной программы, как таблица имен и строка объектного кода, не обязательно должны записываться в смежных областях виртуальной памяти, однако об этом известно только контроллеру памяти. В этом можно убедиться, рассмотрев операции, описанные в табл. 10.2. Например, при генерировании таблицы имен транслятор посылает запрос типа AG контроллеру памяти. Последний возвращает адрес первого слова новой строки. Когда транслятор готов к записи в память управляющего слова таблицы имен (первого слова), он запрашивает выполнение операции SA и передает контроллеру памяти адрес, полученный в результате предшествующей операции AG, и данные, подлежащие записи в память. После обработки данного запроса контроллер памяти передает транслятору адрес следующего слова строки. Для продолжения записи в память таблицы имен транслятор посылает запросы типа SA, передавая в контроллер адрес, полученный после выполнения предыдущей операции SA.

Таблица 10.2. Операции контроллера памяти

Название операции	Назначение и выполняемая функция
AG	Используется для начала новой строки Возвращает адрес первого слова строки
SA	Используется для записи данных в строку Записывает указанные данные по указанному адресу и возвращает адрес следующего слова строки
FF	Используется для просмотра строки Возвращает величину, находящуюся по указанному адресу, и адрес следующего слова строки
FR	Используется для просмотра строки в обратном направлении Возвращает величину, находящуюся по указанному адресу, и адрес предшествующего слова строки
FL	Используется для просмотра строки Возвращает значение и адрес слова, находящегося за указанным словом строки
IG	Используется для включения строки в другую строку
SI	Используется для включения строки в другую строку
SO	Используется для записи данных в строку Запись указанной величины по указанному адресу
DS	Используется для удаления строки
DE	Используется для удаления части строки Изымает все слова строки, расположенные после указанного адреса
FD	Используется для выборки содержимого слова, находящегося по указанному адресу в основной (реальной) памяти
SD	Используется для записи данных по указанному адресу основной реальной памяти
DL	Используется регенератором памяти для добавления списка страниц виртуальной памяти к списку свободных страниц
RG	Используется регенератором памяти для передачи группы слов из списка групп с «информационным мусором» в список доступных для распределения групп слов

Другим примером является посылка центральным процессором контроллеру памяти запросов типа FF для выборки машинных команд. В ответ на каждый такой запрос контроллер памяти производит возврат 64-битового слова (содержащего обычно две команды) и адреса следующей команды. Если в текущей команде содержится адрес операнда, то центральный процессор использует этот адрес при выдаче запроса типа FF для получения управляющего слова идентификатора операнда.

Поскольку адреса, передаваемые контроллеру памяти, на предыдущем этапе им же формируются и передаются произво-

дающим обращение к памяти процессорам, последним не известно, что слова строки логической памяти могут располагаться и не в смежных областях виртуальной памяти. В действительности процессоры не «рассматривают» адреса данных в традиционном смысле; для них адрес — это просто 24-битовое имя некоторой величины, хранимой в памяти.

ВИРТУАЛЬНАЯ ПАМЯТЬ

Функцией контроллера памяти является отображение логической памяти в едином виртуальном адресном пространстве объемом 16 млн. слов. Для того чтобы понять, как осуществляется этот сравнительно сложный процесс, рассмотрим вначале полную структуру виртуальной памяти, изображенную на рис. 10.1.

Страница 1 физической памяти используется для размещения системных таблиц. Страницы 2—4 содержат информацию о каждом пользователе системы. В страницах 5 и 6 находятся буферы ввода-вывода, используемые контроллером каналов ввода-вывода. Оставшиеся страницы образуют адресное пространство, предназначенное контроллеру памяти для организации логической памяти. Как показано на рис. 10.1, вследствие применения логической памяти нет необходимости в выделении каждому пользователю непрерывной области виртуальной памяти. Фактически страницы памяти, выделяемые разным пользователям, размещаются в виртуальной памяти произвольным образом, однако разным пользователям никогда не предоставляется память на одной и той же странице.

Формат страницы последней области виртуальной памяти (применяемой для организации логической памяти пользователей) показан на рис. 10.2. Первые четыре слова используются

Таблица 10.3. Описание формата слова связи группы слов страницы

Номера битов	Назначение битов
0—7	Поле «быстрого поиска»
8—31	Адрес следующей группы строки
32	Указание наличия содержимого в поле «быстрого поиска»
33	Указание наличия ссылки на следующую группу
34	Указание наличия ссылки на предыдущую группу
35	Указание наличия свободного пространства памяти
36	Указание на то, что это последняя группа строки
37	Указание на то, что одно из слов группы — адрес
38—39	Не используются
40—63	Адрес предыдущей группы строки

как заголовок страницы, предназначенный прежде всего для организации связи между страницами, входящими в состав различных списков страниц. Следующими 28 словами являются слова связи групп. Они служат для обеспечения представления памяти на логическом уровне. Последние 224 слова образуют область каждой страницы, используемую для хранения данных.

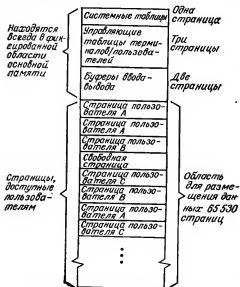


Рис. 10.1. Распределение страниц виртуальной памяти.

Указанные 224 слова доступны пользователю памяти каждой страницы подразделяются на 28 распределяемых областей памяти, состоящих из восьми слов и называемых группами; причем каждая группа имеет свое слово связи группы. Если теперь вновь обратиться к некоторым примерам гл. 9, становится ясно, что, хотя такая строка данных, как таблица имен или строка объектного кода, и не обязательно размещается в смежных областях виртуальной памяти, каждый сегмент строки размером в восемь слов занимает смежные ячейки памяти.

Чтобы понять возникновение принципов логической памяти, рассмотрим описание формата слова связи группы (табл. 10.3). Каждое слово связи группы содержит ссылки в прямом и обратном направлении; этими ссылками являются виртуальные адреса следующей и предшествующей группы в данной строке логической памяти. Говоря иначе, строки логической памяти представляются в виде цепочек слов связи групп и групп слов с данными, относящимися к этим словам связи. Заметим, что адреса, указанные в табл. 10.3, являются 24-битовыми адресами виртуальной памяти. Отсюда следует, что слова связи групп могут ссылаться на группы, находящиеся на других страницах памяти, т. е. строка логической памяти может быть размещена на нескольких страницах.

Чтобы понять, каким образом происходит распределение памяти, предположим, что некоторый процессор, выполняющий обработку для пользователя А, посылает контроллеру памяти запрос на выделение в памяти группы слов (AG). Вначале

контроллер осуществляет поиск страницы виртуальной памяти, предназначенной данному пользователю, в которой имелось бы свободное пространство памяти. (Здесь мы опускаем процедуру поиска контроллером памяти такой страницы. Заметим также, что если такая страница не может быть обнаружена, то контроллер найдет неиспользуемую страницу и назначит ее пользователю А.)

Положим, что страница найдена, в ней уже распределены группы 1, 3 и 5, а остальные группы свободны. Информация об этом регистрируется следующим образом. Заголовок страницы имеет два 5-битовых поля, называемых *началом списка свободных групп* (AGLS — available group list start) и *счетчиком распределенных групп* (IGAC — initial group assignment counter). В данном случае содержимое поля AGLS равно 2, содержимое поля IGAC — 6; в слове связи группы 2 находится виртуальный адрес группы 4, а бит 33 слова связи группы 4 равен 0. Другими словами, свободные группы объединены в список, а в поле IGAC находится номер первой группы из этого списка.

Однако если последние n групп страницы свободны, то они не связываются между собой с целью образования списка (следовательно, и не строится список в описанной выше форме); вместо этого в поле IGAC помещается номер первой из этих групп.

При выполнении операции выделения группы слов памяти (операции AG) контроллер памяти прежде всего выясняет, не содержит ли поле IGAC число 29. Затем он вычисляет виртуальный адрес первого слова группы 6, добавляет 1 к содержимому поля IGAC и производит возврат этого адреса процессору, пославшему запрос. Если бы содержимое поля IGAC равнялось 29, то контроллер взял бы группу из списка свободных групп.

Если следующим запросом является запись данных в память и получение адреса следующего слова (операция SA), то контр-

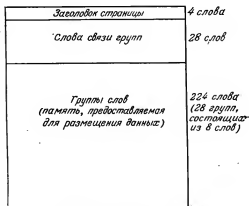


Рис. 10.2. Формат страницы виртуальной памяти.

роллер памяти должен возвратить адрес следующего слова в данной строке. Если заданный в запросе адрес не относится к последнему слову группы, то контроллер просто увеличивает этот адрес на 1 и производит его возврат. После семикратного выполнения операции SA для данной строки контроллер памяти обнаруживает, что данные записываются в последнее слово группы. В этот момент выполняется рассмотренный ранее процесс поиска свободной группы. Полагая, что в результате поиска найдена свободная группа 7, контроллер помещает адрес группы 7 в поле слова связи группы 6, предназначенное для ссылки на следующую группу строки, а адрес группы 6 — в поле слова связи группы 7, предназначенное для ссылки на предыдущую группу строки, и возвращает адрес первого слова группы 7. Обычно при этом контроллер памяти быстро обслуживает семь из восьми запросов, в то время как выполнение восьмого запроса требует существенного дополнительного времени.

Работа контроллера памяти при обслуживании запросов другого типа достаточно понятна. Например, при выполнении операции FF (выборка данных и осуществление возврата адреса следующего слова) контроллер передает указанный в запросе адрес, увеличенный на 1, если выбираемое слово не является последним в группе. Если же это слово последнее в группе, то контроллер должен определить адрес следующего слова строки путем обращения к слову связи данной группы, поиска (с помощью ссылки вперед по списку) слова связи следующей группы и передачи запрашивающему процессору адреса первого слова этой группы.

ВОССТАНОВЛЕНИЕ ГРУППЫ СЛОВ

Если в дальнейшем процессору строка не нужна, он посылает запрос контроллеру памяти на ее освобождение. Процесс *освобождения памяти, занимаемой строкой*, представляет собой операцию, требующую значительного времени. Конечным результатом этой операции является добавление групп слов, в которых находилась строка, в список свободных групп; но выполнить это не так просто, как кажется на первый взгляд. Поскольку строка может быть размещена по нескольким страницам виртуальной памяти (а каждая страница имеет свой собственный список свободных групп слов), строка должна быть разделена на части, относящиеся к отдельным страницам, и группы, в которых размещаются эти части строки, должны быть включены в списки свободных групп каждой страницы. Помимо того что на все это уходит значительное количество машинного времени, может потребоваться еще дополнительное время при обнаружении отсутствия требуемой страницы, так как некоторые

страницы виртуальной памяти могут не находиться в данный момент в основной памяти.

Дополнительные затруднения возникают по той причине, что строка может ссылаться на другие строки, а удаление строки предполагает необходимость изъятия всех строк, связанных с данной строкой. Примером такой ситуации является использование структур языка SPL. Предположим, что A — структура, а строка, представляющая ее в памяти, должна быть исключена. Если некоторые элементы структуры A сами являются структурами, то строка, представляющая A , содержит адреса других строк, последние в свою очередь могут ссылаться на некоторые строки. Например, если элемент $A[1]$ — структура, то первое слово первой группы слов строки, представляющей A , содержит не значение данных, а адрес другой строки.

Если бы указанные выше действия выполнялись контроллером памяти всякий раз при получении им запроса на исключение строки, то контроллер был бы занят чрезмерно длительный интервал времени, буквально приостанавливая работу всей системы. Решением данной проблемы является использование в архитектуре системы отдельного процессора — *регенератора памяти*. Контроллер памяти обрабатывает запрос на исключение строки быстро — благодаря тому, что ему достаточно только добавить эту строку к одному из списков «информационного мусора», т. е. списков использованных, подлежащих удалению строк. Всю же работу по восстановлению памяти выполняет регенератор памяти.

Регенератор памяти имеет самый низкий приоритет доступа к ней; он может посылать запросы контроллеру памяти только тогда, когда последний свободен, другие процессоры не посылают свои запросы контроллеру памяти, и обмен сигналами управления между процессорами не происходит.

Регенератор памяти непрерывно просматривает списки «информационного мусора». При обнаружении содержимого в подобном списке регенератор памяти удаляет первую строку и просматривает все слова связи групп. Регенератор памяти выполняет это, посылая стандартные запросы контроллеру памяти. Регенератор памяти добавляет каждую группу строк к списку свободных групп соответствующей страницы, посылая запросы на освобождение групп (операция RG) контроллеру памяти. Таким образом, контроллер памяти играет подчиненную роль в процессе восстановления памяти, действуя как «подпрограмма» этого процессора.

Если строка содержит адреса других строк, регенератор памяти просматривает каждое слово каждой группы строки, осуществляя поиск не значений данных, а их адресов. При обнаружении подобного адреса адресуемая им строка добавляет-

ся в список строк, подлежащих удалению. Таким образом, память, занимаемая этими строками, будет освобождаться во время последующих просмотров данного списка. (Адреса, подобно строкам данных, сами определяют свой тип; адресам предшествует байт, содержащий код ЕС, указывающий ссылку на структуру, являющуюся элементом данной структуры.)

РАСПРЕДЕЛЕНИЕ СТРАНИЦ ВИРТУАЛЬНОЙ ПАМЯТИ

До сих пор при обсуждении процесса распределения памяти не рассматривался вопрос о том, как контроллер памяти находит свободную страницу. В данном разделе дается ответ на этот вопрос, а также описывается управление страницами виртуальной памяти. Управление страницами подобно управлению группами слов и основано на использовании списков свободных страниц и списков страниц, содержащих «информационный мусор». Одна из системных таблиц в основной памяти содержит заголовок (начальный адрес) списка свободных страниц, в котором находятся все страницы, не выделенные в данный момент пользователю. В заголовке каждой страницы имеется 16-битовое поле, используемое для соединения страниц в единый список. Если страница является свободной, то содержимое поля указывает на связь этой страницы со списком свободных страниц.

Как уже упоминалось, в основной памяти для каждого пользователя имеется управляющая таблица. Все страницы, выделенные в данный момент для обслуживания пользователя, связаны между собой, однако для каждого пользователя предназначен не один, а три списка страниц (TPL1, TPL2 и TPL3). Заголовки этих трех списков находятся в каждой управляющей таблице пользователя.

Целью использования трех списков страниц является минимизация числа страниц, находящихся в основной памяти, путем выделения одной и той же страницы (страниц) для совместно используемых данных, программ и таблиц. В соответствии с принятыми для системы соглашениями список TPL1 включает страницы, в которых находится временная рабочая область пользователя (содержащая, например, текст исходной программы); в список TPL2 входят страницы с таблицами имен программ, стеками, используемыми центральным процессором, и областями для размещения значений переменных программ во время их выполнения; список TPL3 содержит страницы, содержащие объектный код программ пользователя.

Эти три списка показаны на рис. 10.3, для упрощения которого изображены только девять страниц и одна управляющая

таблица пользователя. Рисунок не столь сложен, если рассматривать каждый список отдельно. Страницы 2, 4 и 9 в данный момент не распределены и включены в список свободных страниц. Страница 3 находится в списке TPL1 пользователя А, и, поскольку часть ее памяти свободна, она также входит в список свободных областей памяти страниц, включенных в список

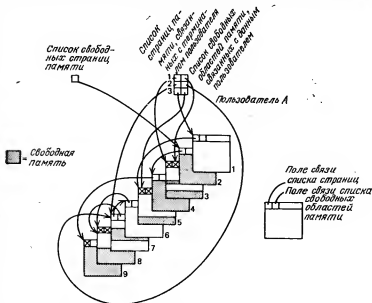


Рис. 10.3. Пример организации списков страниц виртуальной памяти в системе SYMBOL.

TPL1. Страницы 7, 6 и 8 находятся в списке TPL2; в страницах 7 и 8 имеется свободная память, и они включены в список свободных областей памяти, связанный со списком TPL2. Страницы 1 и 5 образуют список TPL3, причем страница 5 включена в соответствующий список свободных областей памяти.

Как указывалось ранее, при обращении к памяти процессор помещает код функции обращения на шесть линий МС основной шины: на четыре линии код, показывающий, какая из 14 возможных операций обращения к памяти подлежит выполнению; на две другие линии код, определяющий, к какому из трех списков страниц памяти пользователя относится данный запрос (этот код используется только при выполнении операций, связанных с распределением памяти).

Предположим, что центральный процессор посылает контрол-

леру памяти запрос на выполнение операции AG (выделить группу слов памяти) и указывает, что этот запрос должен относиться к списку TPL2. Контроллер просматривает список свободных областей памяти на страницах, входящих в список TPL2. Если список свободных областей памяти не является пустым, то распределяется группа слов из страницы, находящейся первой в данном списке. Если же этот список пустой¹⁾, контроллер памяти исключает страницу из списка свободных страниц, добавляет ее в список TPL2 и список свободных областей памяти страниц списка TPL2, помещает 1 в поле IGAC (initial group assignment counter — счетчик количества первоначально распределенных групп слов страницы) и выделяет для запрашивающего процессора группу слов из этой страницы.

ВОССТАНОВЛЕНИЕ СТРАНИЦ

Освобождение (восстановление или регенерация) страниц выполняется регенератором памяти способом, подобным освобождению групп слов. Предположим, пользователь только что закончил компилирование и выполнение программы и намеревается осуществить пуск другой программы. Принимая команду RUN (ПУСК), супервизор системы помещает списки TPL2 и TPL3 пользователя в очередь запросов к регенератору памяти и заполняет нулями заголовки списков TPL2 и TPL3. Регенератор памяти добавляет каждую страницу в список свободных страниц, посылая контроллеру памяти запросы типа DL (см. табл. 10.2).

Освобождая страницу, регенератор заполняет ее нулями, так что распределение и освобождение групп слов страницы не приводит к ее некорректному использованию. Следовательно, нет необходимости в исключении каждой строки страницы списка страниц, подлежащих освобождению. Таким образом, количество запросов на исключение строк, обрабатываемых контроллером памяти, меньше количества запросов на распределение групп слов.

СУПЕРВИЗОР

Супервизор системы — это процессор, выполняющий функции, обычно реализуемые операционной системой. Другие процессоры подчинены супервизору, который указывает им, что делать. Супервизор является в то же время пассивным процессором в том смысле, что его работа инициируется сигналами прерывания, посылаемыми другими процессорами. Супервизор обслу-

¹⁾ То есть на страницах, входящих в список TPL2, нет свободных областей памяти. — *Прим. перев.*

живает эти прерывания, либо обращаясь к другим процессорам, либо сам выполняя требуемую обработку.

Основные функции супервизора системы заключаются в управлении другими процессорами путем обслуживания их рабочих очередей, планирования операций страничного обмена при обнаружении отсутствия в основной памяти требуемой страницы виртуальной памяти и управления системными управляющими таблицами и управляющими таблицами пользователей. Процессом страничного обмена управляет не контроллер памяти, а супервизор, потому что последний имеет более «широкое представление» о функционировании системы и может принимать более эффективные решения, связанные с процессом замещения страниц.

ПРОЦЕСС ОБМЕНА СТРАНИЦАМИ

Во время цикла выдачи ответа на обращение к памяти супервизор системы постоянно контролирует сигналы на линиях МС. Если на этих линиях появляется код 000001 (отсутствие требуемой страницы виртуальной памяти), то супервизор регистрирует адрес памяти, передаваемый по шине в данный момент; таким образом он узнает, какая страница виртуальной памяти должна быть загружена в основную память. Если супервизор системы обнаруживает на этих линиях код завершения 100011, указывающий на успешное выполнение запроса на обращение к памяти и требование к контроллеру памяти исключить страницу из списка свободных страниц, супервизор увеличивает содержимое счетчика числа страниц, предоставляемых в распоряжение данному пользователю. Показания этого счетчика используются для завершения работы тех пользователей, запросы которых на выделение памяти превышают некоторое пороговое значение (например, при закикливании программы, проявляющемся в запросе на выделение чрезмерно большого объема памяти).

Процесс страничного обмена начинается, когда процессор посылает супервизору системы управляющий сигнал, указывающий на отсутствие страницы в основной памяти. Супервизор инициирует выполнение процедуры замещения страниц (реализуемой аппаратными средствами, поскольку супервизор системы не имеет программного обеспечения) и помещает запросы на страничный обмен в очередь к контроллеру памяти на магнитных дисках. Это могут быть запросы на передачу страницы из основной памяти в память на магнитных дисках или из памяти на магнитных дисках в основную память. Затем супервизор просматривает очередь процессора, не обнаружившего в основной памяти нужную страницу, отыскивает в очереди запрос

пользователя и помечает этот запрос как находящийся в ожидании нужной страницы, оставляя его в текущей позиции очереди запросов к данному процессору. Последнему же назначается выполнение другого запроса очереди.

Все дорожки магнитных дисков, используемых в качестве страничной памяти, разделены на четыре сектора. Страница размещается в пределах одного сектора. Каждой странице виртуальной памяти зарезервирована определенная позиция на диске. При вращении диска его контроллер получает информацию о том, какой сектор должен быть следующим под головкой чтения/записи. Контроллер в свою очередь передает эту информацию супервизору системы. Последний использует ее для минимизации потерь времени, связанных с задержкой вращения накопителя, а именно выдает команду контроллеру памяти на обращение к соответствующему элементу его входной памяти.

Существует еще один список страниц, не показанный на рис. 10.3. Это список страниц виртуальной памяти, находящихся в данный момент в основной памяти. Одно из полей заголовка каждой страницы используется в качестве поля связи этого списка. По завершении операции страничного обмена супервизор включает новую страницу в конец данного списка. (Вытесненная из основной памяти страница исключается из списка при выдаче запросов на страничный обмен.) Супервизор завершает процесс удалением отметки об ожидании страницы с запроса пользователя, находящегося в очереди на обслуживание процессором, обнаружившим отсутствие в основной памяти необходимой страницы виртуальной памяти.

ЗАМЕЩЕНИЕ СТРАНИЦ

Алгоритм замещения страниц, реализуемый супервизором системы, представляется достаточно интересным, поскольку, во-первых, он построен по принципу FIFO (first in, first out), в соответствии с которым предпочтение отдается обслуживанию программы, находящейся в начале очереди центрального процессора. Во-вторых, алгоритм использует глобальную информацию о состоянии системы с целью определения страницы, исключение которой из основной памяти в данный момент является наиболее рациональным.

Для каждой страницы, находящейся в основной памяти, супервизор вычисляет *индекс резидентности страницы*, являющийся мерой целесообразности сохранения страницы в основной памяти. Индекс назначается в соответствии с положениями, приведенными в табл. 10.4.

Индекс 0 присваивается в основном страницам, вероятность использования которых в ближайшем будущем мала. Такими

Таблица 10.4. Присвоение страницам значений индекса резидентности

Индекс резидентности страниц	Название и местоположение страниц
3	Страницы, принадлежащие пользователю, запрос на обслуживание которого находится в начале очереди центрального процессора
2	Страницы, принадлежащие другим пользователям, запросы на обслуживание которых находятся в очереди центрального процессора
2	Страницы, содержащие таблицы имен (т. е. страницы, включенные в списки TPL2), принадлежащие пользователям, запросы на обслуживание которых находятся в очереди транслятора
1	Страницы, принадлежащие пользователям, запросы на обслуживание которых находятся в очереди процессора интерфейса ввода-вывода
1	Страницы, не содержащие таблицы имен, принадлежащие пользователям, запросы на обслуживание которых находятся в очереди транслятора
0	Все другие страницы

страницами являются свободные страницы, страницы, принадлежащие пользователям, в работе которых наступила пауза или программы которых только что завершили выполнение, а также страницы пользователей, время обслуживания которых процессором только что превысило допустимый временной интервал.

В процессе замещения страниц супервизор также вычисляет *приоритет загрузки страницы виртуальной памяти пользователя* при обнаружении отсутствия требуемой страницы в основной памяти. Если запрос на обработку, относящийся к пользователю, является первым в очереди центрального процессора, то загрузке страницы присваивается приоритет 3. Если запрос пользователя на обслуживание является первым в очереди транслятора или процессора интерфейса ввода-вывода, то приоритету загрузки страницы присваивается большее из следующих двух значений: 2 или приоритет, присвоенный пользователю (который может принимать значение 0, 1, 2 или 3). Для любых других запросов приоритет загрузки страницы присваивается согласно приоритету пользователя.

Для нахождения страницы, подлежащей удалению из основной памяти, супервизор просматривает очередь страниц, находящихся в основной памяти, от начала до конца (как говорят, сверху вниз). Выбирается та страница, индекс резидентности которой меньше приоритета загрузки. Если такую страницу

найти не удастся, то удаляется первая страница списка, индекс резидентности которой равен приоритету загрузки страницы пользователя, отсутствующей в основной памяти. Если и такой страницы в списке нет, то супервизор отказывается от выполнения запроса на страничный обмен и не помечает находящийся в очереди запрос пользователя как ожидающий вызова страницы. В конечном счете процессор, ранее обнаруживший отсутствие нужной страницы в основной памяти, установит еще раз факт отсутствия той же самой страницы в этой памяти. Однако можно надеяться, что к тому времени состояние системы изменится и требуемую страницу виртуальной памяти можно будет загрузить в основную память.

Итак, алгоритм замещения страниц не произвольно выбирает страницу, подлежащую исключению из основной памяти, а принимает решение, основанное на следующей информации: приоритете задания пользователя, при выполнении которого возникла потребность в вызове страницы виртуальной памяти, а также содержанием каждой страницы и приоритете заданий пользователей, каждому из которых принадлежит страница виртуальной памяти, загруженная в основную память. Пользователь, запрос которого является первым в очереди центрального процессора, обслуживается системой наилучшим образом. Он может «отнимать» страницу у других пользователей (в том числе и свои собственные страницы, использованные во время предыдущих обращений к памяти), однако никакой другой пользователь не может «отнять» у него страницы, расположенные в основной памяти и принадлежащие ему.

УПРАВЛЕНИЕ ОЧЕРЕДЬЮ ПРОЦЕССОРА

Одной из функций супервизора системы является добавление запросов в очередь каждого процессора, их изъятие оттуда и передача сообщения каждому процессору о том, какой запрос из очереди подлежит обработке. Этот алгоритм реализуется аппаратными средствами, однако на него оказывают влияние параметры, хранимые в основной памяти. Указатели начала и конца очереди содержатся в области системных таблиц в основной памяти.

Когда процессор может выполнять новую работу, супервизор указывает ему элемент очереди (запрос), подлежащий обработке, путем просмотра очереди процессора с начала до конца. При этом выбирается запрос, находящийся в очереди первым среди запросов, не ожидающих вызова в основную память страницы виртуальной памяти. Когда запрос помещается в очередь, то для него устанавливается *интервал времени нахождения в очереди в состоянии выполнения*, определяемый значени-

ем соответствующего системного параметра. Супервизор периодически уменьшает величину этого интервала для всех элементов очереди, обслуживаемых процессором. Если для какого-нибудь элемента очереди указанная величина становится равной 0, то элемент перемещается в конец очереди и интервалу времени его выполнения повторно присваивается начальное значение. Таким образом, запрос пользователя обслуживается процессором до тех пор, пока не истечет допустимый интервал времени нахождения его в очереди в состоянии выполнения, после чего вероятность его обслуживания резко уменьшается. Кроме того, новые запросы всегда помещаются в начало очереди для того, чтобы быть уверенными, что они получают (по крайней мере первоначально) приоритет в обслуживании.

Другой системный параметр управляет *глубиной просмотра* супервизором очереди процессора. Если размер очереди превышает значение данного параметра, то запросы пользователей, исчерпавшие предоставленный им промежуток времени, перемещаются в конец очереди и временно исключаются из цикла мультипрограммной работы процессора.

Интервал времени нахождения запроса в очереди в состоянии выполнения предназначен для управления задачами, загружающими работой процессор. Однако это не препятствует задаче страничного обмена монополизировать основную память и процессор, поскольку для подобной задачи характерна тенденция располагаться в начале очереди на чрезмерно длительное время. Поскольку подобная ситуация наиболее часто бывает у центрального процессора, то запрос, находящийся в начале очереди центрального процессора, контролируется системным параметром, определяющим *максимально допустимое время нахождения запроса в начале очереди*. Если период пребывания запроса в начале очереди превышает значение данного параметра, то запрос перемещается в конец очереди.

ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР

Центральный процессор выполняет объектные коды программ на языке SPL. В процессе своей работы он обращается в памяти к адресатам четырех типов: объектной программе, таблице имен, стеку и памяти, используемой для хранения данных. Стек центрального процессора — это список, размещенный в виртуальной памяти; обращение к стеку центральный процессор осуществляет с помощью контроллера памяти.

Часть управляющей таблицы каждого пользователя в нижней области основной памяти используется центральным процессором для хранения информации о состоянии программы пользователя. Наряду с другой информацией центральный про-

цессор хранит в этих управляющих таблицах адрес объектной программы, адрес используемой в данный момент таблицы имен, адрес текущего стека, адрес выполняемой команды и текущее значение параметра LIMIT¹⁾.

ВНУТРЕННИЕ ФОРМЫ ПРЕДСТАВЛЕНИЯ ДАННЫХ

Согласно гл. 9, центральный процессор обрабатывает данные, используя их внутреннее представление, более эффективное для обработки. При обмене данными с другими процессорами (например, с транслятором или процессором интерфейса ввода-вывода) центральный процессор выполняет преобразование формы представления данных из внешней во внутреннюю и обратно. Внешним представлением скаляра, как показано в гл. 9, является строка. Если эти данные не являются числом, то их внутреннее представление подобно внешнему. Если скаляр — число, то он записывается во внутренней форме представления данных, называемой числовой формой.

Числовая форма — это упакованная десятичная запись (две цифры в байте) числа, представляемого в виде значений мантиссы и порядка, т. е. числа с плавающей точкой. Числа записываются в поле, длина которого равна целому числу слов, причем старший бит в последнем байте последнего слова числового поля устанавливается равным 1. Первый байт представляет собой указатель типа данных — тег, определяющий данные как числовые и указывающий знаки порядка и мантиссы. Код F0 в этом байте означает положительные значения мантиссы и порядка, F1 — отрицательное значение мантиссы и положительное значение порядка, F2 — положительное значение мантиссы и отрицательное значение порядка и F3 — отрицательные значения мантиссы и порядка.

Второй байт определяет величину порядка (00—99). Первые четыре бита после последней цифры мантиссы должны соответствовать шестнадцатеричным цифрам F или E: F — обозначает точное значение числа, E — приближенное. Например, число 1283 представляется в следующем виде:

F0 04 12 83 FX XX FX

а число—1234567.891234567EM в виде

F1 07 12 34 56 78 91 00
00 00 23 45 67 EX XX EX

Как показано выше, для представления мантиссы в каждом слове используются только байты с 3-го по 7-й.

¹⁾ Этот параметр определяет точность (число разрядов) представления числовых значений операндов. — *Прим. перев.*

Внутреннее представление структур называется *нормальным*. В такой форме структуры записываются в память в виде одного или нескольких векторов, причем в памяти каждый вектор представлен списком (строкой). Компоненты вектора размещаются в списке последовательно; список завершается словом, начинающимся с управляющего знака «конец вектора» (код F7). Однако если компонента вектора — структура, то она представляется в списке в виде указателя подгруппы структуры. Указатель подгруппы имеет следующий формат:

EC AAAAAA SS BBBBVB

где AAAAAA — адрес вектора, представляющего подгруппу структуры; SS — значение индекса компоненты вектора, последней из тех компонент, к которым производилось обращение, BBBBVB — адрес этой компоненты. (Как показано в табл. 9.3,

Представление
пользователя

101	201	301
102	202	302
103		

Запись на бумаге

<<101|201|301><102|202|302>103>

Управляющее слово
идентификатора

B2	02
----	----

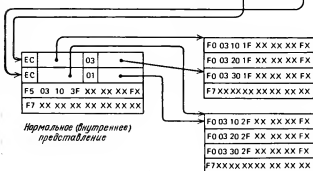


Рис. 10.4. Внутреннее представление структуры.

эти две последние величины — SS и BBBBVB — содержатся также в управляющих словах идентификаторов структур.) Величины SS и BBBBVB используются для быстрого выполнения операций индексирования. Утверждение о достижении повышенного быстродействия этих операций основано на предполо-

жении, что если последнее обращение производилось к i -й компоненте структуры, то следующее обращение будет осуществляться к этому же элементу или к одному из элементов, расположенному в списке после i -й компоненты (например, к $(i+1)$ -й компоненте). При определении индекса компоненты центральный процессор сравнивает значение индекса с величиной SS . Если индекс больше этой величины или равен ей, то поиск требуемой компоненты начинается не с адреса AAAAAA, а с адреса BBBBBB.

Рис. 10.4 иллюстрирует внутреннее представление структуры, имеющей, с точки зрения пользователя, форму «неправильного» массива (таблицы с недостающими элементами). Первый вектор структуры включает три компоненты, две из которых являются указателями подгрупп структуры. Все скаляры представлены в своей внутренней числовой форме. Если предположить, что последним обращением к структуре (назовем ее S) было $S[2, 1]$, а обращением, ему предшествующим, — $S[1, 3]$, то рисунок наглядно демонстрирует использование полей «быстрого поиска», содержащих величины SS и BBBBBB.

ЧЕТЫРЕ ПОДПРОЦЕССОРА

Центральный процессор состоит из четырех подпроцессоров, каждый из которых имеет собственный набор регистров и арифметических устройств. Подпроцессоры подключены к 8-разрядной управляющей шине для передачи между ними управляющих сигналов, а также к основной шине для обмена данными между ними. Подпроцессоры используют основную шину, когда она не занята другими процессорами.

Основным подпроцессором в центральном процессоре является *устройство управления последовательностью выполнения машинных команд*. Это устройство выбирает следующую команду, подлежащую выполнению, и обрабатывает команды с literalными данными, команды перехода и вызова процедур. Для обработки команд другого типа устройство управления последовательностью обращается к остальным подпроцессорам.

Вторым подпроцессором служит *арифметический процессор*, обрабатывающий все арифметические команды и команды сравнения числовых данных. Функциями этого процессора являются также и вычисления с управляемой точностью, для чего используется регистр, содержащий значение параметра LIMIT.

Третий подпроцессор — это *процессор преобразования форматов*, обрабатывающий все команды, которые оперируют скалярами и логическими величинами, и осуществляющий автоматическое преобразование данных из числового представления в строку символов и обратно.

Четвертым подпроцессором является *процессор адресации данных*. Он управляет командами выполнения операций присваивания и обращается к таблице имен по запросам других процессоров, однако его основное назначение состоит в манипулировании структурами. Он ответствен за преобразование формы представления структуры линейной в нормальную и обратное преобразование. Кроме того, именно этот подпроцессор обрабатывает сложные ситуации, такие, как присвоение элементу структуры «значения» другой структуры или скаляра, размер которого превышает размер данного элемента структуры.

Основная часть процессора адресации данных выполняет операции индексирования элементов структуры. Для повышения эффективности выполнения этих операций (исключения обязательного просмотра всех компонент вектора, начиная с первой компоненты) учитывается информация, содержащаяся в полях индексов последних использованных компонент вектора. Однако операции с индексами занимают много времени, поскольку элементы могут иметь различную длину, а их слова могут быть размещены в разных страницах виртуальной памяти. Чтобы избежать просмотра данным процессором каждого слова строки в поисках требуемого слова, используется 8-битовое поле «быстрого поиска» слова связи группы слов страницы (см. табл. 10.3). Эти 8 бит соответствуют восьми словам группы. Единичное значение бита «быстрого поиска» указывает на то, что соответствующее слово является первым словом элемента вектора (компоненты). Следовательно, если процессору адресации данных нужно обратиться к 14-й компоненте, ему понадобится просматривать слова связи групп только до тех пор, пока не встретится 14-й единичный бит «быстрого поиска»; в просмотре каждого слова соответствующих групп нет необходимости.

ТРАНСЛЯТОР

Транслятор — это процессор, транслирующий (компилирующий) исходную программу, составленную на языке SPL, в соответствующие ей объектную программу и одну или несколько таблиц имен. Как упоминалось в гл. 8, этот процесс выполняется не программными, а аппаратными средствами (посредством последовательностных логических схем).

Транслятор функционально разделен на две секции: генерирования объектного кода и формирования таблицы имен. Первая секция просматривает исходную программу и создает соответствующую объектную программу. Когда в процессе этого просмотра встречается буква, не принадлежащая литеральным

данным (она может быть первой буквой идентификатора или зарезервированного слова), управление передается второй секции. Секция формирования таблицы имен продолжает просмотр символов этого слова (имени) языка SPL, пока в нем не встретится ограничитель. После этого она просматривает таблицу зарезервированных слов, хранящуюся в основной памяти. Если обнаруженное в тексте программы имя представляет собой зарезервированное слово, секция формирования таблицы имен генерирует соответствующий код операции.

Если обнаруженное имя не является зарезервированным словом, то анализируется содержимое текущей таблицы имен. Если в этой таблице имеется данное имя, то в секцию генерирования объектного кода передается адрес соответствующего управляющего слова идентификатора. В противном случае в таблицу имен добавляется это имя и формируется новое управляющее слово идентификатора, адрес которого передается в секцию генерирования объектного кода.

После того как таблицы имен и объектные коды всех блоков исходной программы сформированы, выполняется процесс разрешения внешних (глобальных) ссылок. Секция формирования таблиц имен просматривает таблицы имен, осуществляя поиск глобальных идентификаторов. При обнаружении такого идентификатора осуществляется обращение к таблице имен внешнего блока посредством указателя этого блока, расположенного в управляющем слове данного блока, после чего глобальные идентификаторы связываются с соответствующими идентификаторами внешних блоков программы.

Второй процесс установления связей относится к обработке неразрешенных ссылок на процедуры. Если обнаруживается имя вызываемой процедуры, отсутствующей в тексте исходной программы, то осуществляется поиск этой процедуры в системных библиотеках или личных библиотеках пользователей. Поскольку программы хранятся в библиотеках в исходном виде, процесс трансляции должен быть продолжен; при этом создаются новые таблицы имен, а к ранее сформированному объектному коду программы дописывается дополнительный объектный код.

ОСТАЛЬНЫЕ ПРОЦЕССОРЫ

Из общего набора процессоров системы SYMBOL не были рассмотрены следующие: процессор интерфейса ввода-вывода, передающий данные между виртуальной памятью и буферами ввода-вывода, выполняющий запрашиваемые с терминала операции редактирования текста и управляющий операциями ввода-вывода, инициируемым объектными программами; контрол-

лер каналов ввода-вывода, организующий передачу данных между низкоскоростными устройствами ввода-вывода (например, терминалами) и буферами ввода-вывода; контроллер памяти на магнитных дисках, управляющий передачей данных между основной памятью и устройствами внешней памяти; процессор обработки машинных сбоев.

Назначение и принципы функционирования этих процессоров достаточно просты, поэтому их описание может быть опущено.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Как отмечалось выше, система SYMBOL может функционировать без программного обеспечения. Однако опыт работы с системой показал целесообразность разработки средств программного обеспечения. Объем требуемого программного обеспечения оказался относительно небольшим по сравнению с программным обеспечением других подобных систем, но существенно выше предполагаемого вначале.

Одним из первых пакетов программного обеспечения системы SYMBOL был редактор текста, разработка которого обусловлена недостатками редактирования, выполняемого аппаратно процессором интерфейса ввода-вывода. Последний обеспечивает режим редактирования, требующий применения специальных терминалов, реализует слишком примитивный набор функций редактирования и сложен в использовании. Поскольку никакие аппаратно реализуемые функции редактирования не могут служить как функции встроенных блоков программы-редактора текста, большая часть процессора интерфейса ввода-вывода не используется при работе системы с программой-редактором.

Вторым основным пакетом программного обеспечения был ориентированный на язык программирования отладчик. Команда SOURCE-POINTER (указание адреса выполняемого оператора исходной программы), рассмотренная в гл. 9, никогда не использовалась в системе SYMBOL. Эта команда была включена с целью установления связи программных ошибок с соответствующим оператором исходной программы. К сожалению, она оказалась непригодной к использованию и не обеспечивающей результаты, адекватные предъявленным требованиям.

Ее непригодность связана с тем, что возможны изменения (редактирование) исходной программы после выполнения и повторное выполнение текущего объектного кода программы без новой трансляции. Редактирование приводит к записи новой версии исходной программы в другую область виртуальной памяти. Таким образом, команды SOURCE-POINTER в ранее сформированной объектной программе будут задавать ошибоч-

ные ссылки. Неадекватность результата работы команды предъявляемым требованиям проявилась в том, что указывался только ошибочный оператор исходной программы, но не определялись более точно операция и операнд, связанные с данной ошибкой.

Одной из наиболее важных функций программы-отладчика является декомпилирование объектной программы. При обнаружении центральным процессором ошибки в программе управление автоматически передается отладчику, который, анализи-

*** EXECUTION ERROR (ZERO DIVISOR - CODE 20)
IN THE FOLLOWING STATEMENT:

stl ← qmin * (m1 * m2 / (gamma / rho) / (minterm / (1-f)));

RIGHT OPERAND:

| 00 |

LEFT OPERAND:

minterm: | -5.663 |

MONITOR NOW IN CONTROL

?

Рис. 10.5. Результаты декомпилирования объектного кода, выполненного отладчиком.

руя объектный код программы, определяет в нем местоположение команды, с которой связана ошибка, и соседних команд и без обращения к исходному тексту программы преобразует (декомпилирует) машинные команды в оператор языка SPL. Определяя границы искомого фрагмента объектного кода по местоположению окружающих его команд END-OF-STATEMENT (конец оператора исходной программы) и используя таблицу имен, отладчик в большинстве случаев может точно воссоздать оператор языка SPL, за исключением пробелов и необязательных круглых скобок. Затем отладчик выводит на терминал описание ошибки; декомпилированный оператор; стрелку, указывающую на операцию или операнд, связанный с ошибкой, и значения операндов операций. Рис. 10.5 демонстрирует данные, выводимые отладчиком на терминал.

Кроме этого, отладчик позволяет отображать (на экране дисплея, бумаге и т. п.) значения переменных, выполняемые в данный момент операторы языка SPL, а также оператор, вызвавший текущую процедуру.

Сравнительная сложность отладочных функций и простота их реализации в системе SYMBOL позволяют сделать вывод о важной особенности машины с уменьшенным семантическим разрывом между языком программирования и их архитектурой. Отладка программ в этом случае упрощается, кроме того, су-

ществению снижается стоимость разработки отладочных средств, ориентированных на язык программирования. Для сопоставления с другими системами рассмотрим трудности процесса отладки программ и производства средств их отладки у машин низкого уровня ориентации на язык программирования, использующих оптимизирующий компилятор, который осуществляет значительную реорганизацию программы в процессе генерирования объектного кода. В этом случае довольно сложно описать соотношения между представлениями программы на языке высокого уровня и машинном языке, что в свою очередь еще более затрудняет попытки восстановления операторов исходной программы и локализации в них ошибок по имеющемуся в распоряжении объектному коду этой программы.

Другие средства программного обеспечения (из числа разработанных для системы SYMBOL) реализуют учетные функции, выполняемые в начале и конце сеансов связи через терминалы, обслуживают библиотеки подпрограмм и системы файлов. Все компоненты программного обеспечения выполняются центральным процессором и конкурируют с другими программами пользователей за эксплуатацию ресурсов системы.

Системные программы написаны на так называемой усеченной версии языка SPL. Компилирование с такого языка является единственной привилегией и отличием системных программ от программ пользователей. Большинство «усеченных» операторов позволяет программе непосредственно запрашивать операции контроллера памяти (операции, перечисленные в табл. 10.2). Примерами «усеченных» операторов являются AG (Адресация группы), FF (Выборка и получение адреса следующего слова) и SA (Запись и получение адреса следующего слова).

ОБОБЩЕННАЯ ХАРАКТЕРИСТИКА СИСТЕМЫ SYMBOL

Архитектура и реализация системы SYMBOL имеют важное значение во многих отношениях, хотя этой системе и присущи некоторые недостатки. Двумя наиболее важными особенностями системы являются применяемые в ней способы мультипроцессорной обработки и управления памятью. Мультипроцессорная организация системы SYMBOL уникальна в том смысле, что вместо множества одиотипных процессоров общего назначения она предусматривает использование специализированных процессоров, выполняющих разные функции вычислительной системы. Общеизвестным фактом является то, что специализированный процессор может быть разработан быстрее и с меньшими затратами, чем процессор общего назначения, решающий ту же задачу. Специфика механизма управления памятью за-

ключается в высокой степени абстракции представления памяти, обеспечиваемого контроллером памяти. Объекты данных с более сложной организацией, такие, как стеки, очереди, строки переменной длины и списковые структуры, к которым обращаются другие процессоры, легко создаются с помощью операций над логической памятью, тем самым уменьшается общий объем аппаратных средств управления памятью, необходимых системе.

Другая важная особенность системы SYMBOL заключается в том, что в ней в действительности воплощены принципы машины языка высокого уровня. Располагая языком SPL и средствами отладки высокого уровня, пользователь может и не знать внутренней организации системы.

Полюстью аппаратная реализация системы, с одной стороны, является ее достоинством, а с другой стороны, создает определенные трудности. Аппаратная реализация супервизора и транслятора обеспечивает феноменальную быстроту «трансляции, загрузки и выполнения» программы и большую скорость вызова отсутствующих в основной памяти страниц виртуальной памяти. Принцип логической памяти был бы неосуществим при попытке реализовать его программными средствами. Это демонстрирует возможность реализации «на кристалле», т. е. аппаратным путем, функций, традиционно выполняемых с помощью программного обеспечения. Однако полностью аппаратная реализация системы порождает и ряд серьезных проблем, и в частности большую стоимость внесения в систему изменений и расширения реализуемых ею функций. Кроме того, разработчики выяснили, что при проектировании, основанном на аппаратной реализации функций вычислительной системы, так же не избежать ошибок, как и при разработке соответствующего программного обеспечения. Однако представляется возможным достижение некоторых целей проектирования и разрешения некоторых проблем благодаря использованию процессоров с микропрограммным управлением.

Если говорить об архитектуре центрального процессора, то для системы SYMBOL в значительной мере удалось сократить семантический разрыв между архитектурой аппаратных средств и языком программирования. В архитектуре центрального процессора реализованы самоопределяемые данные, данные переменной длины, десятичная арифметика (см. гл. 4). Существенным недостатком архитектуры, как указано в гл. 9, является неэффективное использование памяти машинными командами. Отмечалось также, что аппаратно реализуемый транслятор генерирует объектный код, структура которого далека от оптимальной.

Изучение производительности центрального процессора показывает, что он весьма эффективен при выполнении сложных

операций и совершенно неэффективен при выполнении чаще встречающихся простейших операций. Например, операцияращения значения переменной может потребовать до 12 обращений к памяти. Это обусловлено отсутствием адресации в командах обращения к стеку, косвенной адресацией посредством таблицы имен и размещением стека в памяти.

Основной проблемой, препятствующей использованию системы SYMBOL в качестве ЭВМ общего назначения, является то, что эта система связана с одним языком программирования, причем новым и незнакомым большинству программистов. Одно из решений данной проблемы могло бы заключаться во включении в систему нескольких процессоров, выполняющих функции трансляторов (например, трансляторов для языков SPL, ФОРТРАН, КОБОЛ), однако это представляется неосуществимым из-за сильной ориентации центрального процессора на язык SPL. Другим возможным решением является использование — в добавление к трансляторам для каждого языка — нескольких центральных процессоров, каждый из которых был бы ориентирован непосредственно на определенный язык программирования высокого уровня.

УПРАЖНЕНИЯ

10.1. Почему в основную шину включены линии МТ? Другими словами, почему при выдаче запросов на обращение к памяти в контроллер памяти передается информация, идентифицирующая терминалы и пользователей?

10.2. Центральный процессор часто посылает запрос на выполнение операций выборки из памяти и просмотра строки в обратном порядке. С какой целью это делается?

10.3. При распределении новых групп слов памяти контроллер памяти использует значение счетчика числа первоначально распределенных групп (содержимое поля IGAC), что не является более быстрой процедурой, чем выделение групп из списка групп, доступных для распределения. Для чего в этом случае используется поле IGAC?

10.4. Почему слова связи групп слов должны быть связаны между собой? Почему в слово связи группы помещается ссылка на предыдущую группу списка?

10.5. Почему для обслуживания регенератора памяти в контроллере памяти предусмотрены специальные операции DL и RG (см. табл. 10.2), хотя функция регенератора могла бы быть реализована и с помощью других операций контроллера памяти, например SO?

10.6. Должен ли регенератор памяти просматривать каждую строку для нахождения адресов других строк?

10.7. Каким образом регенератор памяти выделяет адреса среди значений данных в процессе просмотра строк?

10.8. В чем состоит различие между свободной страницей, свободной областью памяти и списками свободных групп слов?

10.9. Как изменится нормальная (внутренняя) форма представления структуры в результате выполнения оператора присваивания $A[2,2] \leftarrow \langle 1234 | 4321 \rangle$?

10.10. Почему адреса первых байтов команд и строк имеют вид $xx2x$ (см. рис. 9.3 и 9.4), а, например, не $xx0x$?

ЧАСТЬ IV

АРХИТЕКТУРА, ОРИЕНТИРОВАННАЯ НА ГРУППУ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

ГЛАВА II

ВЫЧИСЛИТЕЛЬНАЯ СИСТЕМА B1700 ФИРМЫ BURROUGHS

При разработке вычислительной системы общего назначения (т. е. вычислительной системы, обеспечивающей использование не одного, а нескольких языков программирования) с архитектурой, ориентированной на язык программирования, наиболее существенная проблема состоит в том, что значительное сокращение семантического разрыва архитектуры с определенным языком программирования приводит к резкому возрастанию семантического разрыва этой архитектуры с другими языками. Эту проблему иллюстрирует рис. 11.1, на котором представлен граф, отражающий семантические различия между языками и архитектурами. Пусть семантический разрыв между языком программирования и архитектурой машины, реализующей средства этого языка, пропорционален длине пути, который следует пройти по графу между точками, представляющими язык и архитектуру машины. Если некоторая архитектура А1 ориентирована на язык КОБОЛ, то семантический разрыв между этой архитектурой и языком КОБОЛ уменьшается, однако семантический разрыв между этой же архитектурой и другими языками может оказаться больше, чем при использовании традиционной архитектуры фон Неймана.

Два решения этой проблемы представляются очевидными. Первое заключается в создании архитектуры, которая отображалась бы на графе (рис. 11.1) справа от точки представления архитектуры фон Неймана, но не настолько далеко от нее, чтобы оказаться ориентированной в сильной степени на какой-нибудь определенный язык. Другими словами, подобная архитектура должна соответствовать тому, что есть общего в семантике языков программирования. Второе решение предполагает построение системы, реализующей одновременно несколько архитектур (например, за счет использования нескольких процессоров или динамической перестройки архитектуры системы),

каждая из которых ориентирована на определенный язык программирования. Примером последнего подхода в указанной проблеме является семейство ЭВМ В1700 фирмы Burroughs.

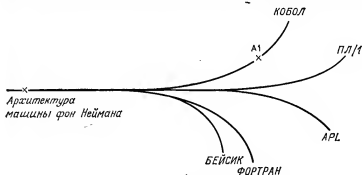


Рис. 11.1. Графическое представление семантических разрывов между языками программирования и архитектурой вычислительных машин.

АРХИТЕКТУРА СИСТЕМЫ В1700 ФИРМЫ BURROUGHS

В отличие от систем, рассмотренных в предыдущих главах, ЭВМ В1700 представляет собой реальную вычислительную систему, используемую для решения экономических задач. По производительности эта машина относится к классу малых и средних ЭВМ, предоставляя в распоряжение пользователя языки программирования БЕЙСИК, ФОРТРАН, КОБОЛ и РПГ/2. Система В1700 — это семейство шести совместимых процессоров, отличающихся друг от друга быстродействием и объемом памяти. Две более поздние разработки подобных систем — В1800 и В1900 — имеют такую же архитектуру, но в них включена мультиобработка и повышено быстродействие схемной логики и обращения к памяти. Система В1700 является мультипрограммной вычислительной системой с виртуальной памятью и может применяться для телеобработки и работы с базами данных.

Ознакомление с архитектурой системы В1700 лучше всего начать с анализа ее работы в мультипрограммном режиме. Предположим, что в стадии выполнения в системе находятся программа на КОБОЛе и программа на ФОРТРАНе, причем в данный момент функционирует операционная система В1700, называемая МСР (Master Control Program — Главная управляющая программа). Последняя определяет, выполнение какой прикладной программы должно быть возобновлено.

Главная управляющая программа написана на языке SDL (System or Software Development Language — Язык разработки программного обеспечения), подобном языкам АЛГОЛ и ПЛ/1. В поисках архитектуры, оптимальной для выполнения программы МСР, разработчики системы создали специальную архитектуру, ориентированную в высокой степени на язык SDL. Следовательно, в данный момент процессор имеет именно такую архитектуру.

Разработчики сознавали, однако, что архитектура, ориентированная на язык SDL, плохо подходит для выполнения прикладных программ (поскольку существует большой семантический разрыв между этой архитектурой и программами, написанными на языках БЕЙСИК, ФОРТРАН, КОБОЛ и РПГ). Для решения данной проблемы была разработана система, использующая несколько архитектур, каждая из которых ориентирована на определенный язык программирования и реализуется отдельной микропрограммой. Если Главная управляющая программа устанавливает, что за определенный интервал времени должно быть возобновлено выполнение программы на языке КОБОЛ, то она указывает реализующим ее аппаратным средствам на необходимость переключения микропрограммы (в данном случае на включение микропрограммы, ориентирующей систему на язык КОБОЛ). Эта операция полностью изменяет архитектуру вычислительной машины, обеспечивая работу процессора с совершенно другим набором команд и другими формами представления данных.

Теперь машина возобновляет выполнение программы на языке КОБОЛ, прошедшей на предыдущем этапе стадию компилирования в объектную программу, которая соответствует архитектуре, ориентированной на язык КОБОЛ. Объектный код, соответствующий каждой подобной архитектуре системы, называется S-языком. Предположим, что произошло прерывание ввода-вывода. Текущая микропрограмма переключается на микропрограмму, ориентированную на язык SDL, и начинает выполняться Главная управляющая программа. Если эта программа решает, что должно быть возобновлено выполнение программы на языке ФОРТРАН, то, установив факт компилирования программы на предыдущем этапе и тем самым преобразования ее в программу на соответствующем (другом) S-языке, она указывает текущей микропрограмме на необходимость переключения на микропрограмму, ориентированную на ФОРТРАН. В результате архитектура системы оказывается ориентированной на язык ФОРТРАН.

Итак, система B1700 динамически изменяет свою архитектуру во время диспетчеризации различных прикладных программ, осуществляемой Главной управляющей программой. Для каж-

дого языка программирования, кроме КОБОЛа и РПГ, система В1700 располагает отдельным S-языком. Поскольку КОБОЛ и РПГ семантически подобны, программы, написанные на этих языках, транслируются на один и тот же S-язык.

РЕАЛИЗАЦИЯ АРХИТЕКТУРЫ СИСТЕМЫ

Хотя вопросы проектирования процессоров преднамеренно исключены из данной книги, сама природа архитектуры системы В1700, ориентированной на группу языков программирования, порождает ряд проблем, связанных с созданием процессора, реализующего эту архитектуру. Поэтому целесообразно хотя бы кратко рассмотреть решение этих проблем.

Одной из проблем является выбор оптимального размера слова памяти и разрядности трактов передачи данных, поскольку эти параметры обычно тесно связаны с размерами полей представления данных в машине определенной архитектуры, а система В1700 объединяет в себе несколько архитектур различного назначения. Например, если архитектура процессора ориентирована на язык ФОРТРАН, оптимальная длина слова составляет 18 или 36 бит, однако такой размер может оказаться слишком большим при ориентации архитектуры на КОБОЛ, поскольку в этом случае желательно иметь машину с байто-ориентированным представлением десятичных чисел.

Решение данной проблемы заключалось в обеспечении динамического изменения основных характеристик процессора при изменении его архитектуры. Например, в системе предусмотрено динамическое изменение разрядности АЛУ процессора. Если архитектура процессора ориентирована на язык ФОРТРАН, то АЛУ работает с данными, представляемыми 18-битовыми словами. Когда архитектура процессора перестраивается на язык КОБОЛ, то данные представляются как 8-битовые слова или слова, длина которых кратна 8 бит. Динамическое изменение разрядности осуществляется с помощью внутреннего регистра, в который микропрограмма записывает требуемую разрядность АЛУ. Если содержимое этого регистра равно 16, АЛУ выполняет операции над 16-битовыми операндами. Физически разрядность АЛУ фиксированна (24 бит). Однако указанный выше регистр маскирует ненужные АЛУ разряды, создавая впечатление, что оно имеет динамически изменяемую разрядность.

Такая же проблема имеет место при работе с памятью; в одних случаях требуется адресация памяти словами длиной 18 бит, в других случаях — словами длиной 8 бит. Решение проблемы в данном случае состояло в обеспечении возможности адресации к каждому биту памяти и указании длины адресуемых полей в интерфейсе, связывающем процессор с памятью.

Если, например, микропрограмме необходимо адресоваться к 16-битовым словам памяти, то при запросе разрешения на запись данных в память эта микропрограмма указывает адрес соответствующего бита, устанавливает длину адресуемого поля (слова) равной 16 бит и передает 16-битовые данные.

Еще одной проблемой создания процессора, реализующего архитектуру В1700, является большой объем управляющей памяти, требуемой для хранения нескольких микропрограмм. Однако, во-первых, эта проблема является не такой серьезной, как это кажется на первый взгляд, поскольку основная часть каждой микропрограммы (исключая средства отладки) занимает ~28 000 бит памяти (менее чем 2000 16-битовых микрокоманд). Во-вторых, разработчик системы располагает несколькими возможностями. Имеется микрокоманда пересылки данных из основной в управляющую память, что позволяет хранить микропрограммы в основной памяти и вызывать их страницы в управляющую память по мере необходимости. Кроме того, можно организовать выборку микрокоманд процессором непосредственно из основной памяти, хотя машина в этом случае будет работать медленнее вследствие большего времени доступа к основной памяти. В системе В1800 отсутствует управляющая память; вместо этого необходимые сегменты микропрограммы вызываются отдельными страницами в высокоскоростную кэш-память. Модели малой производительности систем В1700 и В1800 не имеют ни управляющей памяти, ни кэш-памяти; микрокоманды выбираются процессором из основной памяти.

ПАМЯТЬ И ПРОИЗВОДИТЕЛЬНОСТЬ

Как отмечалось в части I книги, основное преимущество использования архитектуры высокого уровня заключается в существенном уменьшении количества битов, необходимых для представления программы. Это наглядно демонстрирует работа системы В1700. Согласно результатам исследования [1], семь тестовых программ на языке ФОРТРАН занимали 280К байт в системе В1700, 560К байт — в Системе 360 и 450К байт — в системе В3500 фирмы Wiggoughs, т. е. по сравнению с последними двумя системами объем израсходованной памяти в системе В1700 составил 50 и 38% соответственно. Средний объем 20 тестовых программ на языке КОБОЛ составил 450К байт в системе В1700 и 1490К байт — в Системе 360. Таким образом, в данном случае выигрыш равен 70%. Тридцать одна тестовая программа на языке РПГ/2 имела среднюю длину 150К байт в системе В1700 и 310К байт в Системе IBM/3, т. е. экономия памяти составила 52%.

В части I книги также указывалось, что для архитектуры высокого уровня характерно существенное повышение произво-

дительности. Об этом свидетельствуют данные о времени выполнения ориентированной на числовые расчеты программы на языке РПГ: 25 с при использовании системы В1700 и 208 с — для модели 10 Системы IBM/3. В относительных единицах это составит соответственно 8:1 [1]. Однако система В1700 на ~75% дороже. В результате соотношение обобщенных показателей «стоимость — производительность» становится равным 5:1. Хотя среднее время выполнения команды системы В1700 с архитектурой, ориентированной на язык РПГ, равно 35 мкс, а команды Системы IBM/3 составляет 6 мкс, при работе системы В1700 требуется значительно меньшее количество команд (примерно в 50 раз).

ЛИТЕРАТУРА

1. Wilner W. T., Design of the Burroughs B1700, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, NJ, AFIPS, 1972, pp. 489—497.
2. Wilner W. T., Burroughs B1700 Memory Utilization, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, NJ, AFIPS, 1972, pp. 579—586.
3. Wilner W. T., Microprogramming Environment on the Burroughs B1700, Digest of the 1972 IEEE CompCon, New York, IEEE, 1972, pp. 103—106.
4. Burroughs B1700 System Reference Manual, 1057155, Burroughs, Corp., Detroit, 1972.
5. Burroughs B1700 Systems for Computer Science Education, 1077435, Burroughs Corp., Detroit, 1974.

АРХИТЕКТУРА ЭВМ СИСТЕМЫ B1700 ФИРМЫ BURROUGHS, ОРИЕНТИРОВАННАЯ НА ЯЗЫКИ КОБОЛ И РПГ

Поскольку архитектура системы B1700 динамически перестраивается, можно говорить о наличии архитектуры стольких типов, сколько языков программирования допускает к использованию система. Ограничимся здесь рассмотрением архитектуры только одного типа, а именно той, на которую настраивается система для выполнения программ, транслируемых с языков КОБОЛ и РПГ [1].

При создании архитектуры этого типа стремились минимизировать ее семантический разрыв с языками КОБОЛ и РПГ; в результате большинство операторов языка КОБОЛ транслируется в машинные команды «один-к-одному». В соответствии с основными положениями гл. 4 архитектура предполагает использование самоопределяемых (теговых) данных, дескрипторов объектов «составные данные», стеков для управления подпрограммами, ячеек памяти переменной длины и десятичной арифметики. Набор команд процессора, реализующего данную архитектуру, не ориентирован на размещение операндов в стеке или регистрах; для обращения к операндам используется адресация типа «память—память». Адресация лексического уровня не предусмотрена (она была бы бесполезной для программ на КОБОЛе и РПГ); вместо этого память адресуется путем указания имени сегментов памяти и смещения в пределах сегмента—способом, широко используемым в вычислительных системах фирмы Burroughs и являющимся примитивным предшественником потенциальной адресации.

В предыдущих главах анализ архитектуры вычислительных систем сопровождался примерами. Подобная методика неприменима к архитектуре B1700, ориентированной на языки КОБОЛ и РПГ, поскольку в данном случае отсутствует достаточный объем информации об организации памяти и как следствие невозможно проиллюстрировать вид объектной программы, исходный текст которой написан на языке КОБОЛ или РПГ.

ТИПЫ ДАННЫХ

В рассматриваемой архитектуре предусмотрены четыре основных типа данных: строка числовых данных без знака, состоящая из десятичных цифр, каждая из которых представлена 4 бит (4-битовые десятичные цифры); строка числовых данных со знаком, состоящая из 4-битовых десятичных цифр; строка 8-битовых символов без знака и строка 8-битовых символов со знаком. Если строка содержит данные со знаком, то первые 4 или 8 бит представляют знак (+ или —). Когда данные без знака участвуют в арифметических операциях, предполагается, что они представляют собой положительные величины. Если результат арифметической операции описан как данные без знака, то в память записывается его абсолютная величина.

Числовые данные (строки 4-битовых десятичных цифр) всегда имеют определенное числовое значение, представленное в двоично-десятичном коде. Максимальная длина числовых данных — 4095 цифр, не считая знака числа.

Символьные данные (строки 8-битовых символов) представляются в коде EBCDIC. (Они могут также записываться и в коде ASCII, однако обсуждение представления данных в этом коде здесь не приводится.) Если представленные таким образом данные имеют числовые значения (т. е. если все байты строки, за исключением байта знака, содержат двоичный код 1111xxxx, где xxxx — двоично-десятичное представление цифр), то они могут использоваться в качестве операндов арифметических операций. Максимальная длина символьных данных — 1023 символа, не считая знака.

Поскольку используются данные, записываемые в память вместе со своими тегами (указателями типа), обрабатывающие их машинные команды носят универсальный, не зависящий от типа данных характер, причем соответствующее преобразование данных осуществляется автоматически. Например, один из исходных операндов команды ADD (СЛОЖЕНИЕ) может быть строкой из четырех 4-битовых десятичных цифр со знаком, другой операнд — строкой из пяти 8-битовых символов без знака (при условии, что все символы — десятичные цифры), результат сложения — строкой из семи 8-битовых символов со знаком.

Последним типом данных является массив — однородный набор одного из четырех типов данных. Могут использоваться одно-, двух- и трехмерные массивы. Максимальный размер массива числовых данных — 524 287 цифр, массива символьных данных — 262 143 символа. Машина обеспечивает автоматическое индексирование элементов массивов присвоением или вычислением соответствующих индексов — *индексирование при-*

своением и индексирование вычислением. Если одним из операндов команды является массив, то машина автоматически определяет местоположение соответствующего элемента массива. Команда может задавать значения индексов явно (в операндах команды) либо индексы могут быть заданы неявно (находиться в дескрипторе массива).

ПАРАМЕТРЫ ПРОГРАММЫ

Одно из уникальных свойств данной архитектуры состоит в том, что размеры отдельных полей команд и дескрипторов (т. е. адреса памяти) могут изменяться от программы к программе, хотя для любой конкретной программы эти размеры фиксированы. Это позволяет успешно решать наиболее критичную задачу, с которой приходится иметь дело разработчику ЭВМ: определение размера адресных полей. В машинах, архитектура которых предполагает фиксированный размер адресных полей, для одних программ этот размер оказывается излишне большим, и как следствие имеет место непроизводительный расход памяти. Для других программ размер адресных полей недопустимо мал, так что для преодоления этого недостатка требуется применение сложной системы адресации, в Системе 370, например, использование нескольких регистров базы.

В рассматриваемой архитектуре компилятор выбирает минимальные размеры адресных полей, достаточные для записи адресов в конкретной объектной программе. Затем он выполняет трансляцию программы с учетом этих размеров и сообщает их процессору путем запоминания параметров программы вместе с ее объектным кодом. Эти параметры программы, пояс-

Таблица 12.1. Параметры программы

Параметр	Мнемоническое обозначение	Размер поля	Диапазон значений
Длина поля, содержащего имя сегмента данных	PSEG	5	0—18
Длина поля, содержащего смещение данных в сегменте	PDISP	5	1—21
Размер поля, содержащего длину данных	PCELL	5	1—14
Длина индекса дескриптора	PINDEX	5	1—31
Длина дескриптора	PDESC	6	6—57
Длина поля, содержащего смещение адреса перехода, увеличенного на 1	PBDISP	5	2—31
Длина стека подпрограмм	PSSIZE	5	1—31
Смещение нулевой области сегментов данных программы	PDSEGZ	24	Любой
Смещение стека подпрограмм	PSTACK	24	»
Смещение таблицы дескрипторов	PDTAB	24	»

няемые в следующих разделах, перечислены в табл. 12.1, где указаны длина полей, занимаемых каждым параметром, и диапазон допустимых значений параметра. Значения всех параметров задаются положительными двончными числами без знака.

Для иллюстрации использования двух параметров отметим, что адреса памяти состоят из имени сегмента и смещения. Если для конкретной программы $PSEG=00110_2 (6_{10})$, а $PDISP=01100_2 (12_{10})$, то адреса этой программы задаются 6-битовым именем сегмента и 12-битовым смещением. В машинных командах, однако, адреса в таком виде не записываются.

СТРУКТУРА ПАМЯТИ

С каждой объектной программой связана служебная область памяти, показанная на рис. 12.1. Эта область адресуется с помощью регистра базы и содержит информацию о состоянии

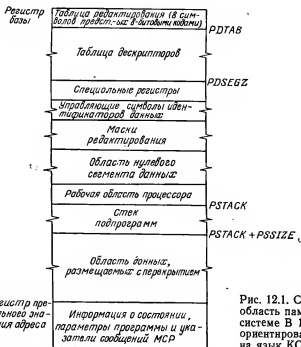


Рис. 12.1. Служебная область памяти в системе В 1700 (архитектура, ориентированная на язык КОБОЛ).

программы и областях хранения данных. Машинные команды и переменные программы размещаются в двух или более отдельных сегментах памяти.

Таблица редактирования содержит восемь специальных символов, используемых при выполнении некоторых команд редактирования. Таблица дескрипторов включает дескрипторы данных для каждой переменной программы. Это аналог таблицы имен, используемой в системе SYMBOL. В области масок редактирования находятся подкоманды редактирования, на которые ссылаются команды редактирования. Область нулевого сегмента данных содержит информацию для управления программой и данные, используемые при взаимодействии прикладной программы с Главной управляющей программой. В стеке

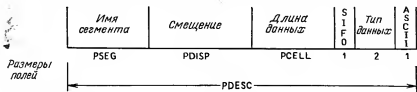


Рис. 12.2. Формат дескриптора скалярных данных.

подпрограмм хранится информация о каждой активной в данный момент подпрограмме.

Таблица дескрипторов, называемая в ЭВМ фирмы Виггоуэкс таблицей COP (current operand table — таблица текущих значений операндов), содержит по одному элементу для каждой переменной программы. Машинные команды обращаются к своим операндам путем указания индексов дескрипторов, находящихся в этой таблице, которые в свою очередь задают местоположение самих данных. Индекс первого элемента таблицы равен 1.

Формат дескриптора скалярных данных (не являющихся элементами массива) показан на рис. 12.2. Первые два поля содержат адрес данных в памяти: имя сегмента, в котором находятся данные, и смещение (выраженное количеством 4-битовых полей) первого бита данных от начала сегмента. Третье поле определяет длину данных без учета знака (если он имеется), задаваемую количеством 4-битовых цифр или 8-битовых символов в зависимости от типа данных. Поля смещения и длины содержат двоичные числа без знака. В этом дескрипторе бит SIF (subscript-index flag) — флажок присваивания/вычисления индекса равен 0, а это означает, что описываемые дескриптором данные скалярного типа.

Содержимое поля типа данных определяет тип данных:

00 — числовые данные без знака;

01 — символьные данные без знака;

- 10 — числовые данные со знаком;
 11 — символьные данные со знаком.

Флажок ASCII указывает, в каком коде представлены символьные данные — в коде ASCII или EBCDIC. Этот флажок влияет на выполнение некоторых машинных команд; в дальнейшем будут рассматриваться данные, представленные только в коде EBCDIC.

Массивы данных описываются расширенным дескриптором, показанным на рис. 12.3. Такие дескрипторы занимают не одно, а несколько слов таблицы дескрипторов. Седьмое по поряд-

Имя сегмента	Смещение	Длина данных	SI F 1	Тип дан- ных	ASCII I 1	Размерность массива	SI 1	Масштаб- ный мно- житель 1	Масштаб- ный мно- житель 2	Масштаб- ный мно- житель 3	Граница массива
PSEG	PDISP	PCELL	1	2	1	2	1	PDISP		PDISP	PDISP
											Пале присутствует в дескрип- торе, если массив трехмерный
											Пале присутствует в дескрип- торе, если массив двумерный

Рис. 12.3. Формат дескриптора массива.

ку поле определяет размерность массива (количество одновременно присваиваемых или вычисляемых индексов элемента массива). Поле SI указывает, какой тип индексирования используется — присваиванием или вычислением. (Различия между указанными выше двумя типами индексирования массивов описываются в документации по языку КОБОЛ.) Если задано индексирование присваиванием, то от одного до трех полей, следующих за полем SI, содержат двоичные величины, представляющие собой масштабные множители, на которые должны умножаться значения каждого индекса для определения местоположения требуемого элемента массива. Поле границы массива содержит смещение последнего элемента массива. Если вычисленное смещение элемента массива оказывается больше, чем величина, находящаяся в этом поле, то генерируется сообщение об ошибке. Например, для массива размером 3×4 ссылка на элемент массива 4,2 не будет восприниматься как ошибочная, а ссылка на элемент 4,4 сопровождается указанием на этот элемент как на ошибочный.

Правила организации индексированных массивов таковы, что за дескриптором массива должны непосредственно следовать от одного до трех дескрипторов каждого присваиваемого или вычисляемого индекса. Эти дескрипторы имеют структуру

дескрипторов скалярных данных. Следовательно, обрабатывая ссылку на дескриптор массива, процессор может определить местоположение текущих значений индексов и использовать эти величины для вычисления адреса текущего элемента массива.

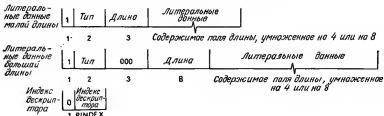
ФОРМАТ КОМАНД

Каждая машинная команда состоит из кода операции ОР (operation code), за которым следует несколько других полей, количество которых определяется типом команды. В этих полях

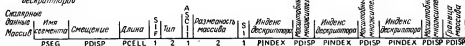
Поле ссылки на дескриптор (DR)



Поле литеpальных данных или ссылки на дескриптор (LIT/DR)



Формат встроенных дескрипторов



Поле адреса перехода (BA)

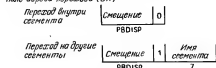


Рис. 12.4. Форматы команд.

обычно находятся литеpальные данные или индексы дескрипторов. Коды операции представляются 3 или 9 бит. Семь наиболее часто используемых команд: INCREMENT—ПРИРАЩЕНИЕ, MOVE ALPHANUMERIC—ПЕРЕСЫЛКА СИМВОЛЬНЫХ ДАННЫХ, MOVE NUMERIC—ПЕРЕСЫЛКА ЧИСЛО-

ВЫХ ДАННЫХ, BRANCH — ПЕРЕХОД, PERFORM ENTER — ВЫПОЛНЕНИЕ ПРОЦЕДУРЫ, COMPARE ALPHANUMERIC — СРАВНЕНИЕ СИМВОЛЬНЫХ ДАННЫХ, COMPARE NUMERIC — СРАВНЕНИЕ ЧИСЛОВЫХ ДАННЫХ — имеют 3-битовые коды операций; коды операций остальных команд записываются с помощью 9 бит в формате 111xxxxxx.

За несколькими ниже рассматриваемыми исключениями поля команды могут содержать ссылку на дескриптор (DR — descriptor reference), литеральные данные либо ссылку на дескриптор (LIT/DR — literal or descriptor reference) или адрес перехода (BA — branch address). Форматы команд показаны на рис. 12.4. Например, в поле DR может находиться индекс элемента таблицы дескрипторов или встроенный дескриптор. Индексы дескрипторов — это двончные числа без знака. Встроенные дескрипторы для скалярных величин имеют тот же самый формат, что и дескрипторы, размещаемые в таблице дескрипторов, тогда как формат встроенных дескрипторов массивов несколько отличен. Вместо того чтобы в этом случае повторять запись дескрипторов значений индексов за встроенным дескриптором массива, в данный дескриптор помещаются индексы дескрипторов полей, в которых находятся дескрипторы значений индексов элементов массива.

Поле, обозначенное как LIT/DR, может содержать литеральные данные или адрес дескриптора. Для литералов, состоящих менее чем из восьми цифр или символов (не считая знака), используются укороченная форма представления, а литералов большей длины — расширенная форма. Литералы могут быть числовыми или символьными данными со знаком или без знака.

Поле, обозначенное как BA (адрес перехода), используется в командах перехода. Если осуществляется переход на выполнение объектного кода, находящегося в текущем сегменте, то применяется формат, соответствующий переходу внутри сегмента. При этом величина смещения — это двончное число без знака, задающее смещение двончного представления команды относительно начала сегмента.

МАШИННЫЕ КОМАНДЫ

В рассматриваемой архитектуре предусмотрено выполнение 38 команд, которые по функциональному назначению могут быть объединены в следующие группы: арифметических операций, пересылки, управления и другого назначения. Семь команд арифметических операций обеспечивают обработку операндов любого из четырех возможных типов данных; при необходимости выполняется автоматическое преобразование данных и при-

соединение ведущих нулей. Если исходными операндами команд арифметических операций являются символьные данные, то их представление в коде EBCDIC должно соответствовать числовой величине (например, с точки зрения выполнения арифметических операций коды F0F5 (5) и 43F3F2 (+32) представляют корректные данные, а код A1BC — некорректные).

Одиннадцать команд пересылки данных перемещают данные из одного поля в другое, выполняя в то же время явно или неявно заданные операции редактирования. Шестнадцать команд управления используются для изменения последовательности выполнения машинных команд. Четыре команды последней группы команд предназначены для обращения прикладной программы к Главной управляющей программе (например, для выполнения операций ввода-вывода).

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Имя команды. INCREMENT (двухадресная операция сложения)

Выполняемая операция. Первый операнд добавляется ко второму.

Формат. OP, LIT/DR, DR

Имя команды. ADD (трехадресная операция сложения)

Выполняемая операция. Суммируются значения двух операндов и результат помещается в поле, адресуемое третьим операндом.

Формат. OP, LIT/DR, DR, DR

Операнды. Первые два операнда могут задавать одну и ту же строку цифр или символов, однако третий операнд должен быть ссылкой на другую строку.

Имя команды. INCREMENT-BY-ONE

Выполняемая операция. Приращение величины, адресуемой операндом, на 1.

Формат. OP, DR

Имя команды. SUBTRACT

Выполняемая операция. Первый операнд вычитается из второго, и результат помещается в поле, адресуемое третьим операндом.

Формат. OP, LIT/DR, DR, DR

Имя команды. DECREMENT

Выполняемая операция. Уменьшение на 1 величины, адресуемой операндом.

Формат. OP, DR

Имя команды. MULTIPLY

Выполняемая операция. Умножение двух операндов и запись результата операции в поле, адресуемое третьим операндом.

Формат. OP, LIT/DR, DR, DR

Операнды. Третий операнд должен быть определен как числовые данные со знаком или без знака.

Имя команды. DIVIDE

Выполняемая операция. Деление второго операнда на первый. Частное записывается по адресу третьего операнда, остаток — по адресу второго операнда.

Формат. OP, LIT/DR, DR, DR

Операнды. Знак остатка совпадает со знаком делимого (второго операнда). Попытка деления на нуль приводит к установке признака переполнения.

КОМАНДЫ ПЕРЕСЫЛКИ ДАННЫХ**Имя команды. MOVE ALPHANUMERIC**

Выполняемая операция. Первый операнд пересылается в поле, адресуемое вторым операндом. Если первый операнд представляет собой числовые данные, значение пересылаемых данных преобразуется в код EBCDIC

Формат. OP, LIT/DR, DR

Операнды. Если длина второго операнда (принимающего поля) больше длины первого операнда (посылающего поля), то передаваемые данные дополняются справа пробелами (если это символьные данные) или нулями (если это числовые данные). Если длина посылающего поля больше длины принимающего поля, то производится усечение передаваемых данных справа. Если в посылающем поле находятся числовые данные, то каждая передаваемая десятичная цифра записывается в принимающем поле в виде 1111xxxx, за исключением знака. Если имеется знак, он преобразуется в соответствующее представление в коде EBCDIC.

Имя команды. MOVE NUMERIC

Выполняемая операция. Первый операнд пересылается по адресу второго операнда.

Формат. OP, LIT/DR, DR

Операнды. При необходимости выполняется надлежащее преобразование данных. Если второй операнд (принимающее поле) определен как данные без знака, то знак первого операнда (посылающего поля) игнорируется. Если принимающее поле имеет большую длину, чем посылающее поле, то передаваемые данные дополняются слева нулями, если меньшую — то производится усечение передаваемых данных слева.

Имя команды. MOVE SPACES

Выполняемая операция. Поле, адресуемое операндом, заполняется пробелами, представленными в коде EBCDIC (0100 0000₂)

Формат. OP, DR

Операнды. Операнд должен быть определен как символьные данные без знака.

Имя команды. MOVE ZEROS

Выполняемая операция. Поле, адресуемое операндом, заполняется нулями.

Формат. OP, DR

Операнды. Если операнд определен как данные со знаком, то записывается знак положительного числа.

Имя команды. MOVE TRANSLATE

Выполняемая операция. Первый операнд пересылается по адресу второго операнда с перекодировкой каждого символа с помощью специальной таблицы перекодировки.

Формат. OP, LIT/DR, DR, DR

Операнды. Первый и второй операнды должны быть определены как символьные данные без знака. Третий операнд указывает таблицу перекодировки. Перекодировка осуществляется следующим образом: числовое значение каждого передаваемого символа, умноженное на 8, используется в качестве смещения относительно начала таблицы перекодировки. В принимающее поле пересылается символ, находящийся в таблице перекодировки по вычисленному таким образом адресу. Если принимающее поле имеет большую длину, чем посылающее поле, то данные в принимающем поле дополняются справа пробелами; в противном случае производится усечение справа.

Имя команды. SCALED MOVE NUMERIC

Выполняемая операция. Числовые данные из поля, адресуемого первым операндом команды, пересылаются в поле, адресуемое вторым операндом. Предварительно к длине посылающего поля добавляется или из нее вычитается масштабный множитель.

Формат. OP, LIT/DR, DR, V, SCL

Операнды. Поле V содержит единственный бит. Поле SCL содержит двоичную величину без знака, количество битов которой равно параметру PCELL. Если V=0, то содержимое поля SCL (масштабный множитель) добавляется к длине посылающего поля, причем предполагается, что расширенное таким образом посылающее поле дополнено справа нулями. Значение масштабного множителя не должно превышать длину посылающего поля. Если V=1, то содержимое поля SCL вычитается из длины посылающего поля. Значение масштабного множителя

не должно превышать длину поля — источника данных. После модификации с помощью масштабного множителя длины пересылаемых данных команда выполняется так же, как и команда MOVE NUMERIC.

Имя команды. CONCATENATE

Выполняемая операция. Содержимое каждого из указанных в команде посылающих полей пересылается в принимающее поле.

Формат. OP, N, DR, LIT/DR1,...,LIT/DRN

Операнды. N — 4-битовое двоичное число без знака, указывающее количество посылающих полей. Следующее поле команды содержит адрес дескриптора принимающего поля. Остальные N полей команды определяют посылающие поля. Данные пересылаются в соответствии с правилами выполнения команды MOVE ALPHANUMERIC.

Имя команды. EDIT

Выполняемая операция. Первый операнд пересылается по адресу второго операнда; пересылка осуществляется под управлением строки подкоманд, находящейся по указанному адресу.

Формат. OP, LIT/DR, DR, DADDR

Операнды. DADDR — двоичное число без знака (с числом битов, равным параметру PDISP), определяющее смещение строки подкоманд редактирования относительно начала нулевой области сегментов данных; в табл. 12.2 приведены эти подкоманды, размещаемые в области «маски редактирования», показанной на рис. 12.1. Если длина адресата операции превышает длину поля — источника данных, адресат дополняется слева нулями. Если длина адресата меньше длины поля — источника данных, производится усечение данных слева. Посылающее поле может содержать данные любого типа, однако адресат должен быть определен как символьные данные без знака.

Подкоманды. Коды подкоманд команды EDIT приведены в табл. 12.2, где pppp — 4-битовое двоичное число, используемое в качестве счетчика повторений. Например, величина 0000 означает, что повторения не производится, т. е. операция выполняется только один раз; величина 0010 указывает на повторение выполнения подкоманды дважды. xxxx — 4-битовое двоичное число без знака, указывающее количество пропускаемых позиций в поле адресата. Величина 0000 указывает на то, что пропуска позиций нет. уuuу — индекс элемента таблицы символов редактирования (табл. 12.3). Если имеет место соотношение уuuу=1010, то это означает, что символ редактирования записан непосредственно за данной подкомандой, а за ним следует очередная подкоманда.

Таблица 12.2. Подкоманды редактирования

Код подкоманды	Назначение
0000nnnn	Пересылка цифр (MOVE DIGITS)
0001nnnn	Пересылка символов (MOVE CHARACTERS)
0010nnnn	Пересылка с подавлением ведущих полей (MOVE SUPPRESS)
0011nnnn	Заполнение символами подавления ведущих нулей (FILL SUPPRESS)
0100xxxx	Смещение по полю адресата в обратном направлении (SKIP REVERSE DESTINATION)
0101yyyy	Безусловное включение символа редактирования (INSERT UNCONDITIONALLY)
0110yyyy	Включение символа редактирования, если установлен признак отрицательного числа (INSERT ON MINUS)
0111yyyy	Включение символа редактирования, если установлен признак подавления ведущих нулей (INSERT SUPPRESS)
1000yyyy	Включение символа редактирования и пересылка цифры или символа данных (INSERT FLOAT)
1001yyyy	Включение символа редактирования (END FLOAT MODE)
1010 0000	Завершение редактирования, если пересылается не нуль (END NONZERO)
1010 0001	Завершение редактирования (END OF MASK)
1010 0010	Установка признака подавления ведущих нулей (START ZERO SUPPRESS)
1010 0011	Формирование дополнения признака защиты данных (COMPLEMENT CHECK PROTECT)
Другие коды	Не определено

Во время выполнения операции редактирования процессор использует три переключателя: S (*знак*), Z (*подавление ведущих нулей*) и P (*защита данных*). В начале выполнения команды EDIT переключатели Z и P установлены в 0. Переключатель S устанавливается в 0, если посылающее поле содержит положительную величину (или величину без знака), в противном случае этот переключатель устанавливается в 1.

Имя подкоманды. MOVE DIGIT

Выполняемая операция. Переключатель Z устанавливается в 1. Очередная цифра или символ посылающего поля пересылается в следующую позицию поля адресата. Если пересылается цифра (4-битовые данные), то производится ее преобразование в код EBCDIC. Если пересылается символ (8-битовые данные), то его первые 4 бит в поле адресата устанавливаются равными 1111.

Имя подкоманды. MOVE CHARACTER

Выполняемая операция. Переключатель Z устанавливается в 1. Очередная цифра или символ посылающего поля пересылается

Таблица 12.3. Стандартные символы редактирования

Номер элемента (индекс) таблицы редактирования	Символ, включаемый в поле адресата команды редактирования
0	+
1	—
2	*
3	.
4	,
5	\$
6	0
7	Пробел
8	+ или —
9	Пробел или —
10	Встроенный 8-битовый символ

в следующую позицию поля адресата. Если пересылается цифра, то производится ее преобразование в код EBCDIC.

Имя подкоманды. MOVE SUPPRESS

Выполняемая операция. Если очередная цифра или символ посылающего поля не представляют собой 0 или если значение переключателя Z равно 1, то выполняется операция пересылки цифры. В противном случае в следующую позицию поля адресата записывается пробел, если значение переключателя P равно 0, или знак звездочки (*), если значение переключателя P равно 1.

Имя подкоманды. FILL SUPPRESS

Выполняемая операция. Если значение переключателя P равно 0, в следующую позицию поля адресата записывается пробел; в противном случае — знак звездочки. В счетчике цифр или символов посылающего поля приращения не производится.

Имя подкоманды. SKIP REVERSE DESTINATION

Выполняемая операция. Величина pppp вычитается из счетчика, указывающего очередную позицию поля адресата. Содержимое счетчика цифр или символов посылающего поля приращения не получает.

Имя подкоманды. INSERT UNCONDITIONALLY

Выполняемая операция. Указанный символ редактирования, находящийся в таблице символов редактирования или являющийся встроенным, записывается в следующую позицию поля

адресата. Если в подкоманде номер элемента таблицы редактирования равен 8, то при $S=0$ записывается знак «+», а при $S=1$ — знак «—». Если номер элемента таблицы редактирования равен 9, то в следующую позицию поля адресата при $S=0$ записывается пробел, а при $S=1$ — знак «—». Содержимое счетчика позиций посылающего поля приращения не получает.

Имя подкоманды. INSERT ON MINUS

Выполняемая операция. В следующую позицию поля адресата записывается определенный символ редактирования, выбираемый с учетом приведенных ниже условий. Содержимое счетчика позиций посылающего поля приращения не получает.

$S=1; T=0, \dots, 7$	Запись символа, находящегося в таблице редактирования под номером T
$S=0; P=0$	Запись пробела
$S=0; P=1$	Запись знака «*»
$S=1; T=8$	Запись знака «—»
$S=1; T=9$	Запись знака «—»
$S=1; T=10$	Запись встроенного символа редактирования

Имя подкоманды. INSERT SUPPRESS

Выполняемая операция. В следующую позицию поля адресата записывается определенный символ редактирования, выбираемый с учетом приведенных ниже условий. Содержимое счетчика позиций посылающего поля приращения не получает.

$Z=1; T=0, \dots, 7$	Запись символа, находящегося в таблице редактирования
$Z=0; P=0$	Запись пробела
$Z=0; P=1$	Запись знака «*»
$Z=1; S=0; T=8$	Запись знака «+»
$Z=1; S=1; T=8$	Запись знака «—»
$Z=1; S=0; T=9$	Запись пробела
$Z=1; S=1; T=9$	Запись знака «—»
$Z=1; T=10$	Запись встроенного символа редактирования

Имя подкоманды. INSERT FLOAT

Выполняемая операция. В следующую одну или две позиции поля адресата записываются указанный символ редактирования и(или) производится пересылка очередной цифры или символа посылающего поля. Выполнение операции зависит от следующих условий (SRC — очередная цифра или символ посылающего поля):

Z=1	Пересылка цифры
Z=0; SRC=0; P=0	Запись пробела
Z=0; SRC=0; P=1	Запись знака «*»
Z=0; SRC≠0; T=0, ..., 7	Запись символа, находящегося в таблице редактирования под номером T; затем пересылка цифры
Z=0; SRC≠0; T=8; S=0	Запись знака «+»; затем пересылка цифры
Z=0; SRC≠0; T=8; S=1	Запись знака «-»; затем пересылка цифры
Z=0; SRC≠0; T=9; S=0	Запись пробела; затем пересылка цифры
Z=0; SRC≠0; T=9; S=1	Запись знака «-»; затем пересылка цифры
Z=0; SRC≠0; T=10	Запись встроенного символа редактирования; затем пересылка цифры

Имя подкоманды. END FLOAT MODE

Выполняемая операция. В следующую позицию поля адресата записывается определенный символ редактирования, выбираемый с учетом приведенных ниже условий. Содержимое счетчика позиций посылающего поля прращения не получает.

Z=0; T=0, ..., 7	Запись символа, находящегося в таблице редактирования под номером T
Z=0; T=8; S=0	Запись знака «=»
Z=0; T=8; S=1	Запись знака «-»
Z=0; T=9; S=0	Запись пробела
Z=0; T=9; S=1	Запись знака «-»
Z=0; T=10	Запись встроенного символа редактирования
Z=1	Подкоманда не выполняет никакой операции

Имя подкоманды. END NONZERO

Выполняемая операция. Выполнение команды EDIT завершается, если произошла пересылка любой, отличной от нуля цифры или символа посылающего поля. В противном случае выполняется следующая подкоманда.

Имя подкоманды. END OF MASK.

Выполняемая операция. Завершение выполнения команды EDIT.

Имя подкоманды. START ZERO SUPPRESS

Выполняемая операция. Переключатель Z устанавливается равным 0.

Имя подкоманды. COMPLEMENT CHECK PROTECT

Выполняемая операция. Формируется дополнение значения переключателя P.

Имя подкоманды. EDIT WITH EXPLICIT MASK

Выполняемая операция. Первый операнд пересылается в поле, адресуемое вторым операндом; пересылка осуществляется под управлением встроенной в команду строки подкоманд.

Формат. OP, LIT/DR, DR, строка подкоманд

Операнды. Строка подкоманд имеет тот же формат, что и литеральные данные (см. рис. 12.4). Первые 2 бит (поле типа) должны быть равны 01. Если следующие 3 бит (поле длины) не равны 0, они указывают длину следующих за ними литеральных данных (строки подкоманд). Если эти 3 бит равны 0, то следующие за ними 8 бит задают длину литеральных данных (строки подкоманд), расположенных непосредственно за этими битами. К использованию операндов применимы соответствующие правила выполнения команды EDIT.

Подкоманды. Используется тот же набор подкоманд, что и при выполнении команды EDIT. Различие между этими двумя командами состоит в том, что в данном случае подкоманды встроены непосредственно в команду EDIT-WITH-EXPLICIT MASK, тогда как при использовании команды EDIT подкоманды записываются отдельно от команды.

Имя команды. MICR EDIT

Выполняемая операция. Первый операнд пересылается в поле, адресуемое вторым операндом. Определенные символы исключаются (не пересылаются), причем количество передаваемых символов помещается в поле, адресуемое третьим операндом.

Формат. OP, RD, DR, DR

Операнды. Команда предназначена для обработки данных при использовании устройства считывания знаков, нанесенных «магнитными чернилами»; детали выполнения операции здесь не обсуждаются.

Имя команды. MICR FORMAT

Выполняемая операция. Первый операнд пересылается и размещается в определенном формате в поле, адресуемое вторым операндом.

Формат. OP, DR, DR

Операнды. Команда предназначена для обработки данных при использовании устройства считывания знаков, нанесенных «магнитными чернилами»; детали выполнения операции здесь не обсуждаются.

КОМАНДЫ УПРАВЛЕНИЯ

Имя команды. BRANCH

Выполняемая операция. Управление передается команде с указанным адресом.

Формат. OP, BA (см. рис. 12.4)

Имя команды. COMPARE ALPHANUMERIC

Выполняемая операция. Каждый бит первого операнда сравнивается с соответствующим битом второго операнда. Если выполняется заданное в команде условие, то управление передается команде, находящейся по указанному адресу.

Формат. OP, LIT/DR, DR, R, BA

Операнды. R представляет собой 3-битовое поле, задающее следующие операции сравнения операндов: 001 (>), 010 (<), 011 (\neq), 100 (=), 101 (> или =) и 110 (< или =).

Имя команды. COMPARE NUMERIC

Выполняемая операция. Первый операнд сравнивается со вторым как числовые величины. Если выполняется заданное в команде условие, то управление передается команде, находящейся по указанному адресу.

Формат. OP, LIT/DR, DR, R, BA

Операнды. Поле R используется так же, как и в команде COMPARE ALPHANUMERIC.

Имя команды. COMPARE REPEAT

Выполняемая операция. Производится последовательное сравнение битов первого операнда с битами сегментов второго операнда. Если выполняется заданное в команде условие, то управление передается команде, находящейся по указанному адресу.

Формат. OP, LIT/DR, DR, R, BA

Операнды. Оба сравниваемых операнда должны быть определены как символьные данные без знака. Длина поля, адресуемого вторым операндом, должна быть кратна длине поля, адресуемого первым операндом. Поле R используется так же, как и в команде COMPARE ALPHANUMERIC.

Имя команды. COMPARE FOR SPACES

Выполняемая операция. Операнд сравнивается бит за битом с содержимым поля, заполненного одними пробелами. Если вы-

полняется заданное в команде условие, то управление передается команде, находящейся по указанному адресу.

Формат. OP, DR, R, BA

Операнды. Операнд должен быть определен как символьные данные без знака. Поле R используется так же, как и в команде COMPARE ALPHANUMERIC.

Имя команды. COMPARE FOR ZEROS

Выполняемая операция. Операнд сравнивается с содержимым поля, заполненного нулями, по правилам сравнения числовых величин. Если заданное в команде условие выполняется, управление передается команде, находящейся по указанному адресу.

Формат. OP, DR, R, BA

Операнды. Поле R используется так же, как и в команде COMPARE ALPHANUMERIC.

Имя команды. COMPARE FOR CLASS

Выполняемая операция. Анализируется формат операнда. Если он соответствует формату, заданному в команде, управление передается команде, находящейся по указанному адресу.

Формат. OP, DR, C, BA

Операнды. Содержимое 2-битового поля C задает требуемую проверку принадлежности операнда к определенному классу данных: 00 (только буквы), 01 (только цифры), 10 (не только буквы), 11 (не только цифры). Операнд должен быть определен как символьные данные.

Имя команды. BRANCH ON OVERFLOW

Выполняемая операция. Проверяется наличие признака переполнения, и при его обнаружении управление передается команде, находящейся по указанному адресу.

Формат. OP, V, BA

Операнды. V — 1-битовое поле. Переход выполняется, если $V=1$ и признак переполнения установлен в 1 либо $V=0$ и признак переполнения установлен в 0.

Имя команды. SET OVERFLOW

Выполняемая операция. Явная установка в 1 или сброс признака переполнения.

Формат. OP, V

Операнды. V — 1-битовое поле. Содержимое этого поля присваивается биту признака переполнения. (Признак переполнения устанавливается в 1 неявным образом в результате деления на 0.)

Имя команды. GO TO DEPENDING ON

Выполняемая операция. Управление передается по одному из нескольких адресов в зависимости от значения операнда.

Формат. OP, DR, N, BA0, BA1,...,BAN

Операнды. N — 10-битовое двоичное число без знака. Если операнд меньше 1 или больше N, то в качестве адреса перехода используется содержимое поля BA0. Если значение операнда принадлежит диапазону от 1 до N, то операнд используется в качестве индекса для определения соответствующего адреса перехода.

Имя команды. ALTER

Выполняемая операция. В указанную область памяти записывается адресная константа.

Формат. OP, DADDR, BA

Операнды. DADDR — двоичное число без знака (с числом битов, равным параметру PDISP), определяющее смещение адресуемой области относительно начала нулевого сегмента данных служебной области программы; начало этого сегмента задается значением параметра PDSEGZ. Содержимое поля BA (адрес перехода) записывается в указанную область памяти.

Имя команды. ALTERED GO TO PARAGRAPH

Выполняемая операция. Управление передается команде, находящейся по указанному адресу. В качестве адреса перехода используется адресная константа, выбираемая из памяти.

Формат. OP, DADDR

Операнды. DADDR — это двоичное число без знака (с числом битов, равным параметру PDISP), определяющее смещение адресуемой области относительно начала нулевого сегмента данных служебной области программы; начало этого сегмента задается значением параметра PDSEGZ. Адрес перехода, записанный в этой области, определяет местоположение команды, которой передается управление.

Имя команды. PERFORM ENTER

Выполняемая операция. Слово, содержащее имя сегмента с текущей командой, смещение следующей команды относительно начала сегмента и константу, загружается в стек подпрограмм; управление передается команде, находящейся по указанному адресу.

Формат. OP, K, BA

Операнды. Поле K содержит 8-битовую константу.

Имя команды. PERFORM EXIT

Выполняемая операция. Если константа, содержащаяся в слове, находящемся на вершине стека, совпадает со значением, заданным в команде, то управление возвращается на выполнение команды, адресуемой этим словом. В противном случае про-

должается выполнение команд, следующих за командой **PERFORM EXIT**.

Формат. OP, K

Операнды. Поле K содержит 8-битовую константу. Если имеет место совпадение констант, то слово, находящееся на вершине стека, изымается из стека.

Имя команды. ENTER

Выполняемая операция. Слово, содержащее имя сегмента с текущей командой и смещение относительно начала сегмента следующей команды, подлежащей выполнению, загружается в стек подпрограмм; управление передается команде, находящейся по указанному адресу.

Формат. OP, BA

Имя команды. EXIT

Выполняемая операция. Управление возвращается команде, указываемой словом, находящимся на вершине стека. Это слово изымается из стека.

Формат. OP

ПРОЧИЕ КОМАНДЫ

Имя команды. CONVERT

Выполняемая операция. Операнд преобразуется в 24-битовое двоичное число без знака и записывается в указанную ячейку нулевого сегмента данных служебной области программы.

Формат. OP, DR, DADDR

Операнды. Операнд должен быть положительным десятичным числом, которое преобразуется в двоичное число и записывается по адресу, задаваемому полем DADDR. Это поле содержит двоичное число без знака (с числом битов, равным параметру PDISP), определяющее смещение адресата операции относительно начала нулевого сегмента данных служебной области программы; начало этого сегмента определяется параметром PDSEGZ.

Имя команды. COMMUN[ICATE

Выполняемая операция. Длина и абсолютный адрес операнда записываются в общий для данной программы и Главной управляющей программы буфер адреса сообщения. Управление передается Главной управляющей программе.

Формат. OP, DR

Операнды. Перед записью в буфер длина операнда преобразуется из двоично-десятичного или символического представления в

двоичное число. Абсолютный адрес операнда в двоичной форме также записывается в буфер.

Имя команды. LOAD COMMUNICATE

Выполняемая операция. Последние 24 бит информации в общем для данной программы и Главной управляющей программы буфере адреса сообщения помещаются в указанную ячейку нулевого сегмента данных.

Формат. OP, DADDR

Имя команды. MAKE PRESENT

Выполняемая операция. Загружается указанный сегмент данных. Адрес области данных (определяемой относительно ее базы) помещается в указанную ячейку нулевого сегмента данных.

Формат. OP, SEG, DADDR

Операнды. Поле SEG, длина которого равна параметру PSEG, указывает имя сегмента данных.

УПРАЖНЕНИЯ

12.1. Какое значение параметра PINDEX (размер поля индексов дескрипторов) должно использоваться транслятором, если для выполнения программы, написанной на языке КОБОЛ, требуется таблица дескрипторов, состоящая из 43 слов?

12.2. Каким должен быть размер команды выполнения двухадресной операции сложения (INCREMENT), если первый операнд не литерал?

12.3. Каким должен быть размер команды INCREMENT, увеличивающий на 2 значение переменной?

12.4. Расшифруйте следующую команду MOVE ALPHANUMERIC (первые 3 бит — код операции; предполагается, что параметр PINDEX=6): 0001000100111001101011

12.5. Что записывается в поле адресата команды, приведенной в п. 12.4, если адресат определен как символьные данные без знака длиной в 3 символа?

12.6. Что записывается в 10-символьное поле адресата команды EDIT, если посылающее поле содержит шестнадцатеричную величину 00001045, а строка подкоманд имеет вид:

10100011 01010101 00100101 01010011 00000001 10100001?

12.7. Какие атрибуты набора команд архитектуры, ориентированной на язык КОБОЛ, являются наиболее оригинальными? Почему?

12.8. Напишите небольшую программу на языке КОБОЛ и выполните ее трансляцию «вручную» (на бумаге) для последующего выполнения этой программы машиной с рассматриваемой архитектурой (т. е. сформируйте дескрипторы, параметры программы и поток машинных команд).

ЛИТЕРАТУРА

1. The material in this chapter is used by permission granted by the Burroughs Corporation. Most of the material is paraphrased from B1700 COBOL/RPG-S-Language, 1058823-015, Copyright 1973, Burroughs Corporation, Detroit, MI.

ЧАСТЬ V

АРХИТЕКТУРА, ОРИЕНТИРОВАННАЯ НА ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

ГЛАВА 13

НАЗНАЧЕНИЕ АРХИТЕКТУРЫ СИСТЕМЫ SWARD

Следующей подлежит рассмотрению архитектура вычислительной системы SWARD (SoftWAre-orientED machine — машина, ориентированная на программное обеспечение). Единственной целью создания такой экспериментальной машины было выяснение возможности разработки архитектуры вычислительной системы, аппаратные средства которой реализуют большую часть функций системы, связанных с организацией и выполнением программ, т. е. функций программного обеспечения машины. Важность этой цели определяется тем, что наиболее существенным недостатком современных вычислительных систем и программных средств является большая стоимость разработки надежных программ. Много усилий затрачено на разработку проектов программного обеспечения, которые позволяли бы ликвидировать этот недостаток. Здесь можно назвать и создание принципов более эффективной разработки программ, совершенствование приемов и методологии тестирования, разработку языков программирования, обеспечивающих написание программ, менее подверженных ошибкам, развитие математических методов доказательства корректности программ. Однако приложенные усилия не привели к существенному сдвигу в направлении решения указанной проблемы. К тому же проблема разработки дешевого и надежного программного обеспечения осложняется следующими дополнительными факторами:

1) повышением требованиями к точности и надежности функционирования программ в новых важных сферах применения программного обеспечения, таких, как управление воздушными и космическими полетами, системы медицинского обслуживания, банковские системы с обработкой данных в оперативном режиме, системы управления в реальном масштабе времени и др.

2) нехваткой квалифицированных программистов, что ощущается сейчас и прогнозируется на ближайшее будущее.

Несмотря на определенные успехи, достигнутые в технологии разработки программного обеспечения, можно отметить и следующие недостатки:

1) производительность труда в программировании не увеличилась сколько-нибудь заметно, и стоимость разработки программ не претерпела изменений, сравнимых с изменением стоимости аппаратных средств вычислительных систем (например, стоимость написания одной команды для микропроцессора обычно превышает стоимость самого процессора);

2) при разработке стандартного программного обеспечения свыше 50% расходов приходится на организацию процедур тестирования и отладки;

3) равное $1/20$ отношение числа ошибок при разработке программного обеспечения и написания прикладных программ к общему числу операторов не является, к сожалению, нетипичным.

В связи с изложенным выше становится ясно, почему была предпринята попытка подойти к решению проблемы с другой стороны — попытаться модифицировать архитектуру машины, являющуюся аппаратной поддержкой средств традиционного программного обеспечения.

Первоначально принципы этой архитектуры были сформулированы в работе [1], а затем описаны в первом издании данной книги. С тех пор рассматриваемая архитектура претерпела существенные изменения [2, 3] и была реализована в аппаратном виде в Институте системных исследований фирмы IBM.

Идея создания архитектуры, повышающей надежность разрабатываемых программ, не является новой, но SWARD — первая система, в которой осуществляется попытка всестороннего охвата подлежащих решению проблем. Предшествующие работы в этом направлении имели значительно более частный характер. Можно, например, отметить работы [4, 5], в которых предлагался вариант машины и архитектуры операционной системы для определения некоторых типов программных ошибок (ошибок адресации). В этой системе любое обращение к памяти выполняется косвенным образом через дескриптор, определяющий положение и размер области памяти, ее содержимое и предоставляемые ей «привилегии» доступа (некоторая разновидность потенциальной адресации). Однако в программах на языках высокого уровня подобные средства защиты могут оказаться полезными только для устранения небольшой части всех возможных ошибок. В работе [6], посвященной архитектуре машин и отладке программ, показаны преимущества вычислительной системы, аппаратные средства которой опознают данные с неопределенными значениями, используют теги, делающие слова памяти самоопределяемыми, проверяют количе-

ство и тип фактических параметров при обращениях к процедурам. В работе [7] отмечаются достоинства средств обнаружения ошибок и отладки программ, используемых машинами с тепловой организацией памяти. Другие исследователи [8] указывают на целесообразность снабжения машины (на уровне аппаратных средств) информацией о всех возможных последовательностях вызова модулей. Это позволит выявлять ошибки, приводящие к неправильной последовательности вызова модулей. Указывалось также на то, что процесс отладки мог бы стать более эффективным, если бы система сохраняла информацию о нескольких последних выполнявшихся командах или переходах в программе [9]. Автор работы [10] высказывает мнение в более общей форме о том, что значительные сдвиги в разработке больших программ могут произойти только при существенных изменениях структуры аппаратных средств (машины), и в частности отмечает преимущества использования одноуровневой памяти.

ЦЕЛИ СОЗДАНИЯ НОВОЙ АРХИТЕКТУРЫ

Знакомство с системой SWARD можно было бы начать с рассмотрения ее архитектуры, однако без определения основных целей ее создания многие характеристики этой архитектуры оставались бы неясными. Поэтому прежде всего проанализируем цели проектирования, приведшие к появлению этой системы.

Побудительным мотивом к поиску новых решений было стремление усовершенствовать процесс разработки программ (системных и прикладных) и повысить их надежность при выполнении. При этом выдвигалось требование принципиального подобия языков программирования, используемых для написания таких программ, тем языкам, которые получили наибольшее распространение в настоящее время. Можно сказать, что цель заключалась в создании более благоприятной среды для программистов и корректных программ, которая в то же время была бы неподходящей для программ с ошибками. Было сформулировано шесть основных требований, которым должна удовлетворять новая система:

- 1) обнаружение семантических ошибок в программах;
- 2) ограничение последствий возможных ошибок в программном обеспечении;
- 3) усовершенствование процесса проектирования программного обеспечения и приемов программирования;
- 4) упрощение процесса разработки программ;
- 5) более эффективное тестирование и отладка за счет специальных средств разработки и сопровождения программного обеспечения;

6) упрощение структуры двух традиционно наиболее сложных видов программного обеспечения — операционных систем и компиляторов.

Для удовлетворения первого требования введена классификация семантических ошибок по 37 различным типам. В отличие от логической ошибки семантическая ошибка характеризуется допущенными в программе нарушениями семантических спецификаций используемого языка. (Заметим, что многие логические ошибки приводят к семантическим.) Было принято решение о том, что в общем случае эти ошибки не могут быть выявлены на этапе компилирования. Однако из этого не следует, что компиляторы принципиально не могут в ряде случаев обнаруживать ошибки такого рода. Дело в том, что компиляторы просто не способны обнаруживать все ошибки каждого из этих типов. Перечислены некоторые основные типы семантических ошибок:

1. Обращение к переменной, значение которой в данный момент не определено (или не установлено). Это может быть простая переменная, элемент массива, часть (поле) записи или структуры, адрес, элемент в цепочке знаков и т. д. Анализ данных указывает на то, что ошибки этого типа являются наиболее распространенными.

2. Обращение к элементу массива с указанием индекса этого элемента, выходящего за пределы данного массива.

3. Обращение посредством переменной, используемой в качестве указателя, ссылки или для разрешения доступа, к части памяти, не предназначенной для хранения данных.

4. Обращение посредством переменной типа «указатель» к части памяти, уже освобожденной программой.

5. Присвоение значению переменного типа или атрибута, отличного от того, который ожидает компилятор. Например, программа читает данные из файла и обращается к отдельным полям каждой записи в соответствии с определением последней, в то время как фактическое представление данных в записи может оказаться несоответствующим определению записи.

6. Количество фактических параметров, передаваемых процедуре, не равно ожидаемому количеству, т. е. числу ее формальных параметров.

7. Типы передаваемых фактических параметров отличаются от типов соответствующих формальных параметров процедуры.

8. Несоответствие определений глобальных переменных в разных модулях.

9. Данные, адресуемые переменной типа «указатель», не имеют атрибутов, которые ожидает компилятор (например, указателю, используемому в программе на языке ПЛ/1 для базирования одной структуры данных, может быть присвоен адрес

другой структуры, а затем этот указатель попытаются использовать для обращения к данным первой структуры).

10. Модификация процедурой входного фактического параметра (значение которого изменению не подлежит).

Немаловажным является вопрос о том, сколь значительно число разнообразных ошибок, действительно имеющих место в программах. Данные работы [1] позволяют подтвердить обоснованность проведенной классификации ошибок. Так, в выборке из 39 ошибок, допущенных в программном обеспечении решения аэрокосмических задач и задач обработки информации в реальном масштабе времени, 13 ошибок относятся к выделенным здесь типам семантических ошибок. Анализ 3361 ошибок программирования, обнаруженных при работе с большой системой контроля и управления, показывает, что по крайней мере 487 ошибок приходится на перечисленные выше типы ошибок. В упомянутой работе также отмечено, что обнаружение ошибок этих типов требует определенных усилий в том смысле, что не все из них выявляются на первоначальных стадиях отладки.

Хотя зависимости между типом ошибки и сложностью ее отыскания не установлены, тем не менее можно отметить, что ошибки выделенных типов относятся к ошибкам, обнаруживаемым с наибольшими затратами усилий. Частично это объясняется способностью таких ошибок придавать недетерминированный характер работе программы.

Второе требование, предъявляемое к рассматриваемой архитектуре, касается способности системы ограничивать последствия допущения ошибок. Для достижения этого требуется выполнение пяти условий, связанных с системой адресации и защиты памяти от несанкционированного доступа. Одно из них, например, заключается в том, чтобы ошибка в данной программе (прикладной или системной) не вызывала непредусмотренных изменений в другой программе или в обрабатываемых ею данных. Согласно другому условию, системные программы (например, операционная система) не должны обладать возможностью неявной адресации к прикладным программам или обрабатываемым ими данным. Имеется также условие, в соответствии с которым необходимо, чтобы ошибки в программном модуле не могли приводить к изменению данных, не предназначенных для использования данным модулем. Иначе говоря, адресное пространство модуля должно быть ограничено его собственными локальными переменными и параметрами, т. е. должен быть реализован принцип разбения всего адресного пространства на небольшие области санкционированного доступа. Третье требование, предъявляемое к рассматриваемой архитектуре, сводится к обеспечению возможности применения современных прогрессивных методов проектирования и разработки

программ. Перечислим некоторые условия реализации этого требования.

1. Наличие эффективных средств разработки и выполнения программ с большим числом модулей.

2. Отказ от использования глобальных переменных и аналогичных способов адресации данных, например типа неявно заданного совместного использования переменных вложенными процедурами. Отметим, что это требование, по существу, сводится к отказу от использования в архитектуре многоуровневой лексической адресации.

3. Отказ от программирования на языке ассемблера.

4. Возможность работы с абстрактными, определяемыми пользователем типами данных.

5. Ориентация на работу со структурированными программами; наличие в мониторах средств синхронизации обработки программных модулей.

Четвертое требование предполагает упрощение процесса разработки программ благодаря закладываемым в систему принципам ее организации и их влиянию на языки программирования. Данное и предыдущее требования к архитектуре преследуют такую цель, как создание условий для предупреждения ошибок программирования. Большинство условий, удовлетворяющих этим требованиям, были сформулированы в процессе разработки данной архитектуры, а не заблаговременно. Часть этих условий касается упрощенного представления операций ввода-вывода, способов независимого определения данных в программах, более упорядоченных форм сообщений об ошибках и использования простых принципов взаимосвязи программ и данных.

Пятое требование, предъявляемое к архитектуре вычислительной системы, предполагает придание ей таких свойств, которые поддерживают и благоприятствуют функционированию средств тестирования и отладки, снижая прямые и накладные расходы на их проведение. Допустимо, в частности, предположить, что аппаратные средства системы могут способствовать выполнению следующих функций тестирования и отладки:

- 1) трассировки выполнения команд программы;
- 2) трассировки последовательности вызова процедур;
- 3) формирования таблиц частоты выполнения отдельных операторов программы;
- 4) трассировки выполнения команд перехода;
- 5) определения местоположения обнаруженных ошибок и выполнения некоторых действий по их исправлению.

Шестое требование к архитектуре предполагает такую организацию последней, которая позволила бы упростить структуру компиляторов и операционных систем. Это возможно при

сокращении семантического разрыва между языком программирования и архитектурой машины, а также при наличии таких базовых механизмов операционной системы, как механизмы управления и синхронизации параллельных процессов.

СРАВНЕНИЕ СИСТЕМ ПО ИХ СПОСОБНОСТИ ОБНАРУЖИВАТЬ ОШИБКИ

Основная цель, выдвигавшаяся при разработке системы SWARD, заключалась в создании более благоприятных условий для выявления семантических ошибок в программах. Вот почему небезынтересно выяснить, как обнаружение таких ошибок выполняется в других системах. Для целей сравнения было выбрано 22 наиболее распространенных типа семантических ошибок (остальные типы ошибок встречаются крайне редко или характерны только для отдельных языков программирования).

При этом было бы естественно рассматривать те вычислительные машины, которые широко используются в настоящее время. Однако в основном они соответствуют традиционной модели фон Неймана, а поскольку известно, что машинами с такой архитектурой большая часть рассматриваемых типов ошибок не обнаруживается, подобное исследование представляло бы незначительный интерес. Поэтому проведем сравнение машин нестандартной архитектуры, таких, как SWARD, SYMBOL, учебная ПЛ-машина, и Burroughs 6500/6700, Система 38. Последняя система, имеющая двухуровневую архитектуру (см. гл. 4), рассматривается с точки зрения функционирования интерфейса уровня M1, обеспечивающего на этапе выполнения реализацию контроля любого вида, который может быть создан средствами программного обеспечения этого уровня. Для получения более полного представления о перечисленных системах, имеющих нетрадиционную архитектуру, сопоставим их с Системой 370 с традиционной архитектурой. Однако в качестве ее конкретного варианта будем использовать некоторую абстрактную модель, являющуюся комбинацией средств Системы 370 и оптимизирующего компилятора языка ПЛ/1, разработанного фирмой IBM. Такая модель Системы 370 обеспечивает на этапе выполнения реализацию любого контроля, задаваемого средствами компилятора языка ПЛ/1.

Сравнительные данные по указанным системам приведены в табл. 13.1. В графе «Количество типов ошибок, обнаруживаемых полностью» этой таблицы указано количество типов ошибок для каждой системы, которые обнаруживаются во всех случаях. Если система обнаруживает лишь часть ошибок какого-либо типа или если она способна к выявлению этих ошибок

Таблица 13.1. Эффективность обнаружения ошибок различными системами

Вычислительная система	Количество типов ошибок, обнаруживаемых полностью	Количество типов ошибок, обнаруживаемых частично	Количество необнаруживаемых типов ошибок
Абстрактная модель			
Системы 370 и компилятора ПЛ/1	1	3	18
B6500/6700 фирмы Burroughs	6	0	16
Система 38	6	4	12
SYMBOL	10	0	12
Учебная ПЛ-машина	11	2	9
Система SWARD	21	0	1

только по указанию программиста, то данные о таких типах ошибок помещаются в графу «Количество типов ошибок, обнаруживаемых частично». Если же данная система не способна выявлять ошибки некоторого типа, то сведения о соответствующих типах ошибок помещаются в графу «Количество типов необнаруживаемых ошибок».

Если для простоты рассуждения принять, что все 22 типа ошибок имеют одинаковую значимость, то, согласно данным табл. 13.1, среди рассматриваемых систем имеются как очень надежные, так и весьма посредственные в этом отношении. Абстрактная модель Системы 370 из 22 типов ошибок обнаруживает полностью ошибки только одного типа (деление на нуль). Машины с теговой памятью, памятью высокого уровня организации, а также ориентированные на сугубо структурированный характер заданий (например, в виде совокупности процедур), такие, как SYMBOL, учебная ПЛ-машина и SWARD, обладают наилучшими характеристиками. Вычислительная система SWARD не обнаруживает ошибки только одного типа — выход за диапазон допустимых значений переменной. Так, например, при использовании в программе описания вида

I: INTEGER range 1..12;

система SWARD ни на этапе трансляции, ни на этапе выполнения не обеспечивает выявление ошибки при выходе значения переменной I за диапазон допустимых значений (от 1 до 12). Это не следует считать неким упрощением при разработке принципов функционирования системы SWARD; было решено, что ошибки такого рода лучше всего обнаруживать, принимая во внимание стоимостные затраты и расход машинного времени посредством вставляемых в определенные места программы специальных команд тестирования, выполняемых компилятором. Так, машина SWARD располагает командой RANGE-CHECK, предназначенной для проведения этой проверки.

Если для некоторой системы не все 22 типа ошибок оказываются в графе «Количество ошибок, обнаруживаемых полностью», то возможны два варианта решения проблемы выявления необнаруживаемых ошибок. Первый вариант предполагает генерирование компилятором машинного кода контроля наличия подобных ошибок.

Однако стоимость таких средств контроля колеблется от сравнительно умеренных значений (связанных с затратами памяти и машинного времени, в частности при контроле принадлежности индексов элемента массива допустимому диапазону значений) до необычайно больших величин (например, при необходимости выявления переменных с неопределенными значениями или неразрешенных адресных ссылок). Поэтому если средства контроля и реализуются на практике, то не в системах массового промышленного производства, подобных ориентированным на программирование на языке ПЛ/1, а только в учебных компиляторах.

Другой вариант решения упомянутой проблемы сводится к игнорированию невыявленных ошибок, что влечет за собой отказ от полной и строгой реализации требований спецификаций языка программирования, вследствие чего правильная работа системы становится зависимой от программиста.

В заключение остановимся на двух моментах, имеющих отношение к компиляторам. Во-первых, несмотря на то что рассмотренные ошибки в общем случае могут быть выявлены лишь на этапе выполнения программы (поскольку они определяются физическими связями, устанавливаемыми между значениями, адресами и признаками только к этому времени), некоторые признаки этих ошибок могут быть обнаружены и на этапе компилирования. Если такая возможность имеется, то следует предусматривать соответствующие процедуры поиска. В частности, если компилятор может выявить очевидные ссылки на переменные с неопределенными значениями, использование значений индекса, выходящих за установленные пределы, или несоответствие фактических параметров формальным (что можно обнаружить, если обе процедуры компилируются совместно и атрибуты формальных и фактических параметров относятся к типу статических), имеет смысл процедуры поиска ошибок этих типов включить в структуру компилятора. Несмотря на это, по-прежнему остается желательной полная проверка машиной наличия всех возможных ошибок и на этапе выполнения. Это необходимо для выявления тех ошибок, которые не могут быть обнаружены во время компилирования, а также для того, чтобы не ставить работу всей системы в зависимость от правильности и надежности работы всех входящих в систему компиляторов.

Во-вторых, не следует полностью полагаться на решение всех возникающих проблем, которые приводят к ошибкам, средствами описания, предоставляемыми языком программирования. Например, средствами языка можно указать компилятору, что всем переменным, которым присвоение исходных значений программой не предусматривается, должно быть присвоено одно и то же определенное значение (например, нулевое значение для числовых переменных, пробелы для строк символов). Хотя такие действия и устраняют возможность появления ошибок, носящих случайный (недетерминированный) характер, они похожи скорее на благие пожелания, чем на действия, гарантирующие исключение соответствующих ошибок. Конечно, можно пойти по пути сокращения средств языка программирования: уменьшить количество типов данных, запретить независимое компилирование программных модулей, отказаться от средств манипуляции именами или указателями переменных. Однако такие меры могут привести только к значительной потере общности принципов и средств программирования.

ОБЗОР АРХИТЕКТУРЫ СИСТЕМЫ SWARD

В данном разделе описываются основные характеристики архитектуры системы SWARD и их связь с проблемой эффективной разработки программного обеспечения.

ТЕГОВАЯ ПАМЯТЬ

В рассматриваемой архитектуре принят принцип организации памяти с помощью тегов. Такая память называется *теговой*, или *памятью с самоопределяемыми данными*. Благодаря использованию этого принципа машина становится понятно назначение атрибутов операндов команд. Она легко обнаруживает в операциях несовместимые операнды и в процессе выполнения команд осуществляет автоматически требуемое преобразование данных. Данные каждого типа могут принимать значение, называемое «неопределенным»; машина способна обнаруживать попытки использовать данные с неопределенными значениями.

Элементы данных, снабженные тегами и называемые ячейками, имеют переменный размер (длину). Архитектурой системы не предусмотрена фиксация размера слова данных, что позволяет машинным командам адресоваться к ячейкам как к операндам. Благодаря этому модель данных, используемая архитектурой системы, хорошо согласуется с моделями языков программирования.

ВЛОЖЕННЫЕ ТЕГИ

Принцип теговой организации памяти получил в рассматриваемой архитектуре дальнейшее развитие: одни теги могут быть вложены в другие. Это позволяет представлять данные более высокого уровня организации, такие, как массивы и структуры записи. Не программа, а машина берет на себя решение задачи адресации массива и автоматического контроля принадлежности индексов допустимым диапазонам их значений. Вложенные теги используются также для определения базированных переменных в языке ПЛ/1 (переменных доступа в языке Ада) и данных, определяемых пользователем, т. е. так называемых *абстрактных данных*.

Принцип вложения тегов удобно использовать для описания сложных данных. Например, описываемая пользователем четырехкомпонентная запись (т. е. запись из четырех полей) может включать в качестве первого компонента двумерный массив, каждый элемент которого представляет собой запись, и т. д.

ПРИНЦИП ПОТЕНЦИАЛЬНОЙ АДРЕСАЦИИ

В архитектуре системы SWARD применен принцип адресации и защиты памяти от несанкционированного доступа, называемый потенциальной адресацией. Согласно этому принципу, система рассматривается как множество объектов, каждый из которых при создании машиной получает присущее только этому объекту имя. Программы лишены возможности создавать или преобразовывать адреса. Каждое обращение к объекту, после того как он уничтожен, выявляется системой как ошибка.

Потенциальные адреса и объекты используются для создания моделей памяти высокого уровня (а не традиционных моделей низкого уровня) — это была одна из задач, стоявших перед создателями архитектуры системы SWARD. На рис. 13.1 приведен пример возможного состояния памяти системы. Сплошными стрелками показаны ссылки средствами потенциальной адресации; штриховыми — внутренние связи между элементами, поддерживаемые системой. Для данной архитектуры характерны пять типов объектов. Четыре из них (модуль, процесс-машина, порт и память данных) в явной форме создаются и используются программами, а один (запись активации) создается неявным образом в момент вызова модуля.

Потенциальные адреса защищены от несанкционированного использования благодаря тому, что для представления каждого из них применяется одна теговая ячейка из 15 возможных типов таких ячеек (данных). Как показано на рис. 13.1, потенци-

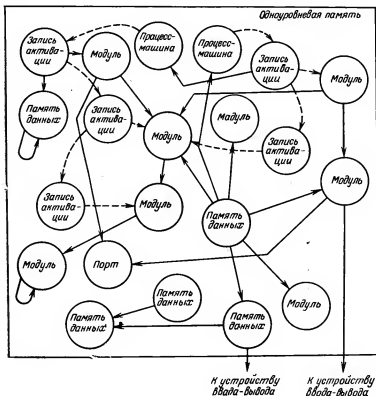


Рис. 13.1. Пример состояния памяти системы.

альные адреса используются также для обращения к устройствам ввода-вывода последовательного действия, не имеющим своей памяти.

ОДНОУРОВНЕВАЯ ПАМЯТЬ

В данной архитектуре понятие виртуальной памяти обобщено настолько широко, что оказывается ненужным понятие вторичной (внешней) памяти. В частности, исключено понятие файла; вместо него в программах используется понятие массива. Как следствие излишними оказались и понятия операций обмена данными ввода-вывода со вторичной памятью. Адресация ко всем данным в системе выполняется единообразно, и все другие понятия архитектуры (например, использование тегов) относятся ко всем данным в равной степени.

Архитектура не предоставляет программные средства для описания запросов на выделение памяти или для описания соот-

ветствующих процессов. Хотя выделение памяти имеет место, оно выполняется машиной без явного запроса со стороны программы. Это происходит, например, в результате вызова очередного модуля, когда машина формирует запись активации для локальных переменных модуля. В этих условиях приходится говорить не о возможностях программ запрашивать память, а о их возможностях запрашивать такие типы ячеек, как строки или массивы (динамическое выделение которых встроено в объект «память данных»).

ОБЛАСТИ САНКЦИОНИРОВАННОГО ДОСТУПА

Каждая программа или процедура представляется в системе объектом «модуль», который содержит последовательность команд, получаемых в результате компилирования, и описание

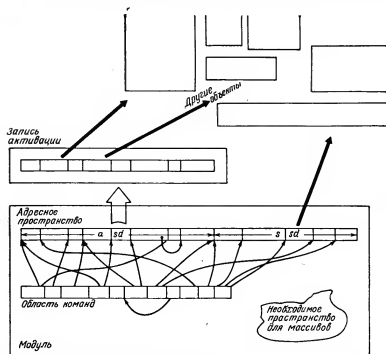


Рис. 13.2. Модуль и его связи.

адресного пространства (указание группы ячеек памяти с тегами). Структура модуля показана на рис. 13.2. Команды некоторого модуля могут адресоваться только к элементам собствен-

ного адресного пространства этого модуля, хотя через параметры и потенциальные адреса эффективно реализуются косвенные ссылки и за пределы этого адресного пространства. Можно, таким образом, сказать, что данная архитектура способствует более полной реализации принципов модульного программирования; ограничивает последствия возможных ошибок; защищает программы, включая системные, от несанкционированного доступа других программ, в том числе от несанкционированного доступа самих к себе.

УПРАВЛЕНИЕ ПОДПРОГРАММАМИ

Предусмотрено несколько команд, реализующих стандартные средства языков высокого уровня для вызова подпрограммы. Так, команда CALL сохраняет информацию о состоянии текущего модуля, создает запись активации для вызываемого модуля и присваивает ей исходное содержимое, переходит на работу с новым адресным пространством и начинает выполнение вызванного модуля. При каждом вызове выполняется также проверка соответствия типов фактических и формальных параметров.

Как показано на рис. 13.2, адресное пространство модуля разделено на две части: *статическую* (ssd — static storage die) и *динамическую* (asd — automatic storage die). Ячейки статической части адресного пространства постоянно закреплены за модулем. Во время каждого вызова точки входа модуля система создает объект «запись активации», куда копируются определения ячеек из динамической части адресного пространства. Когда команда обращается к ячейке динамической части адресного пространства, система автоматически выполняет переадресацию в соответствующую ячейку текущей записи активации.

Модуль может представлять собой процедуру с одной или несколькими точками входа или группу взаимосвязанных процедур (называемых пакетом в языке Ада). Какой из вариантов будет иметь место, зависит от заданного режима работы компилятора и действий программиста.

ИЕРАРХИЧЕСКАЯ ОБРАБОТКА ОШИБОК

В архитектуру системы включен специальный механизм обработки ошибок, имеющий достаточно однородную структуру и ориентированный не столько на обслуживание системы в целом, сколько на обслуживание отдельных процессов. В каждом модуле могут быть заданы точки входа для обработки ошибок и типы обрабатываемых ошибок. На рис. 13.3 изображено функционирование механизма обработки ошибок. При обнаружении ошибки во время выполнения некоторого модуля машина в об-

ратном порядке просматривает список модулей, последовательно вызывавшихся данным процессом. Поиск ведется до тех пор, пока не будет найден первый модуль, «выражающий желание» обработать ошибку данного типа. После нахождения такого мо-

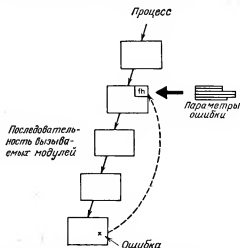


Рис. 13.3. Обработка ошибок в системе SWORD.

дуля машина обращается к нему через его точку входа (аналогично вызову подпрограммы), и передает пять фактических параметров с информацией о типе ошибки и состоянии программы в этот момент времени. Последующие действия определяются логикой программы обработки ошибок этого модуля. Предусмотрены команды для прекращения работы программы обработки ошибок и одна команда для явного задания условий возникновения ошибки.

Таким образом, в общем случае объект типа «модуль» может состоять

из локальных переменных, описателей данных, процедур функционального назначения и процедуры обработки особых случаев.

ПРОЦЕСС-МАШИНА

Одним из пяти типов объектов в архитектуре является объект «процесс-машина». Это абстрактное понятие, служащее для обозначения средств управления параллельными процессами. Создание и уничтожение процесс-машины означает формирование и разрушение программами отдельных процессов. В соответствии с общими принципами архитектуры это понятие означает лишь средство, с помощью которого программы могут формировать «линию своего поведения». Можно отметить, что это понятие не связано с другими понятиями архитектуры, в частности оно не имеет никакого отношения к адресации.

МЕХАНИЗМ ПРИЕМО-ПЕРЕДАЧИ

Для обеспечения связи между процессами в архитектуре системы SWORD предусмотрены специальный абстрактный объект «порт» и две машинные команды SEND и RECEIVE. Команда

SEND определяется почти так же, как и команда CALL, с тем отличием, что при выполнении команды CALL в точку входа модуля передаются управление и список фактических параметров, а при реализации команды SEND через порт поступают фактические параметры. Иначе говоря, команда SEND служит для передачи данных, а не управления. При работе интерфейса приемо-передачи (SEND/RECEIVE), как и при вызове подпрограмм, выполняется проверка типов данных. Как отмечалось выше, обращение к устройствам ввода-вывода без памяти осуществляется средствами потенциальной адресации; операции ввода-вывода выполняются посредством команд RECEIVE и SEND.

КОМАНДЫ, ИНВАРИАНТНЫЕ К ТИПУ ОБРАБАТЫВАЕМЫХ ДАННЫХ

Теговая организация памяти позволяет в рамках данной архитектуры ограничиваться небольшим набором однородных по структуре команд, инвариантных к типу обрабатываемых данных. В частности, для сложения предусмотрено только одна команда ADD, а для пересылки данных в памяти служит тоже одна команда MOVE. Семантика (содержательная часть) команд определяется атрибутами их операндов. Например, команда MOVE может использоваться для перезаписи целого числа в виде вещественного числа с плавающей точкой (имеет место автоматическое преобразование данных), для пересылки строки символов на место, занимаемое другой строкой символов, для записи скаляра во все элементы массива или для пересылки всех элементов одного массива в другой.

МОЩНЫЙ НАБОР КОМАНД

Кроме перечисленных выше команд система SWARD содержит команды адресации и перемещения частей строк в пределах строк, а также команду SEARCH для поиска в массиве элемента с заданным значением. Для синхронизации процессов предназначены команды GUARD и UNGUARD. Их можно использовать для предотвращения одновременного выполнения двумя или большим числом процесс-машин критических участков программ. Эти команды разрабатывались для обеспечения возможностей синхронизации при проектировании специальных программ-мониторов.

СРЕДСТВА КОСВЕННОЙ АДРЕСАЦИИ

В рассматриваемой архитектуре принцип потенциальной адресации был расширен таким образом, чтобы предоставить возможность потенциальному адресу содержать ссылки на другие

потенциальные адреса. Благодаря этому, если программа содержит обращение к потенциальному адресу, машина интерпретирует это как ссылку на последний потенциальный адрес в звене таких адресов. Это свойство можно использовать для введения дополнительных уровней защиты данных, для контроля операционной системой соблюдения санкционированного доступа к отдельным объектам или для динамического замещения объектов (например, модулей), к которым обращается программа во время выполнения.

СРЕДСТВА ТРАССИРОВКИ ПРОГРАММ

Архитектурой предусмотрены команды, подключающие средства трассировки переходов, изменивших последовательность выполнения команд, и переходов, не изменивших этой последовательности, а также трассировки вызовов подпрограмм отдельными модулями. При обнаружении ситуаций, требующих трассировки, система регистрирует их подобно состояниям ошибки и пускает в действие описанный выше механизм обработки ошибок.

ДОПОЛНИТЕЛЬНЫЕ СРЕДСТВА ЗАЩИТЫ ОТ НЕСАНКЦИОНИРОВАННОГО ДОСТУПА

В дополнение к таким средствам защиты, как упоминавшиеся выше потенциальная адресация, области санкционированного доступа и косвенная потенциальная адресация, данная архитектура располагает средством программного ограничения возможностей копирования потенциальных адресов, а также командой для назначения объектам новых имен. Можно заметить, что теговая структура памяти сама по себе является вспомогательным средством защиты доступа при работе с потенциальными адресами: для получения доступа к некоторому объекту требуется не только его потенциальный адрес, но и знание его типа.

ВИРТУАЛЬНЫЙ ХАРАКТЕР МАШИНЫ

Некоторые атрибуты архитектуры системы SWARD, такие, как потенциальная адресация и описание всех процессов в системе в терминах абстрактных объектов, определяют такие ее черты, которые характерны для систем с виртуальной организацией памяти, хотя эта цель при разработке специально не преследовалась. В условиях такой виртуальной среды отдельные программы могут не иметь явных связей с операционной системой, а выполнение программы в машине может поддерживаться одновременно несколькими операционными системами.

ЛИТЕРАТУРА

1. Myers G. J., The Design of Computer Architectures to Enhance Software Reliability, Ph. D. dissertation, Polytechnic Institute of New York, New York, 1977.
2. Myers G. J., SWARD — A Software-Oriented Architecture, Proceedings of the International Workshop on High-Level Language Computer Architecture, University of Maryland, College Park, MD, 1980, pp. 163—168.
3. Myers G. J., Buckingham B. R. S., A Hardware Implementation of Capability-Based Addressing, *Operating Systems Review*, 14(4), 13—25 (1980).
4. Spier M. J., A Model Implementation for Protective Domains, *International Journal of Computer and Information Sciences*, 2(3), 201—229 (1973).
5. Spier M. J., A Pragmatic Proposal for the Improvement of Program Modularity and Reliability, *International Journal of Computer and Information Sciences*, 4(2), 133—149 (1975).
6. Ehrman J. R., System Design, Machine Architecture, and Debugging, *SIG-PLAN Notices*, 7(8), 8—23 (1972).
7. Fesutel E. A., On the Advantages of Tagged Architecture, *IEEE Transactions on Computers*, C-22(7), 644—656 (1973).
8. Kane J. R., Yau S. S., Concurrent Software Fault Detection, *IEEE Transactions on Software Engineering*, SE-1(1), 87—99 (1975).
9. Saal H. J., Shustek L. J., On Measuring Computer Systems by Microprogramming, Microprogramming and Systems Architecture, Infotech State of the Art Report 23, Berkshire, England, Infotech, 1975, pp. 473—489.
10. Dennis J. B., Computer Architecture and the Cost of Software, *Computer Architecture News*, 5(1), 17—21 (1976).

СИСТЕМА SWARD

Рассмотрение системы SWARD, как и других систем, архитектура которых отлична от модели фон Неймана, целесообразно начать с принципов организации ее памяти. В этой главе обсуждаются способы представления данных и адресации в системе SWARD и основные понятия, касающиеся организации памяти. В конце главы приводятся два примера, иллюстрирующие работу системы SWARD в режимах компилирования и выполнения.

При описании архитектуры системы SWARD используются два термина: *система* и *машина*. Первый из них применяется для обозначения всей совокупности технических средств, второй отождествляется с введенным выше абстрактным понятием «процесс-машина». Система представляется как набор «процесс-машин», количество которых меняется динамически на протяжении времени жизни системы. Вопрос о соответствии между процесс-машинами и реальными, аппаратно реализованными процессорами (один процессор используется одной процесс-машиной или один или несколько процессоров принадлежат нескольким процесс-машинам) зависит от конкретной реализации системы SWARD.

ТИПЫ ДАННЫХ

Прежде чем приступить к рассмотрению типов данных, используемых в системе SWARD, необходимо определить несколько основных понятий, связанных с организацией памяти. За единицу исчисления размеров выделяемой памяти принимают *токен* — совокупность четырех смежных битов. Адресуемую единицу памяти называют *ячейкой*. Последняя образуется из некоторого (переменного) количества смежных токенов. В исходной программе каждой ячейке соответствует переменная или константа (данные). Содержимое ячейки состоит из двух частей: *тега*, описывающего атрибуты содержащихся в ячейках данных, и самих *данных*.

В рассматриваемой архитектуре используются ячейки, а следовательно, и данные 15 типов, из них десять называются простыми, а пять — вложенными (с вложенными тегами).

ПРОСТЫЕ ЯЧЕЙКИ

Структура ячеек для простых данных различных типов показана на рис. 14.1. Знак в форме треугольника используется для указания границы между полем, предназначенным для тега, и полем, представляемым для данных ячейки того или иного типа. Как и следовало ожидать, тег однозначно характеризует тип данных и сохраняется неизменным во время выполнения программы, т. е. не может быть изменен ею. В поле данных ячейки находится ее текущее содержимое.

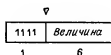
Ячейка «целое число» может содержать целое число в диапазоне от $-8\,388\,607$ до $+8\,388\,607$ либо *признак неопределенности*. Это содержимое ячейки представляется в виде двоичного числа и хранится в двоичном дополнительном коде. Содержимое данной ячейки, равное $800\,000$ в шестнадцатеричной системе счисления, служит признаком неопределенности для обозначения неприсвоенного значения. Такое представление признака неопределенности выбрано потому, что оно соответствует записи максимального отрицательного числа в двоичном дополнительном коде. Следует отметить, что максимальное положительное число в качестве признака неопределенности не используется.

Ячейка «целое число увеличенной разрядности» может содержать целое число в диапазоне от $-2^{47}-1$ до $2^{47}-1$, или примерно в диапазоне $\pm 1,4 \cdot 10^{14}$. Признак неопределенности в этом случае равен $800\,000\,000\,000$ (в шестнадцатеричной системе счисления).

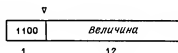
Ячейка «порядковое число» содержит числовую величину от 0 до 254 в двоичной форме. Текущее содержимое ячейки является элементом упорядоченной числовой последовательности и может быть использовано как целое положительное число. Величина 11111111 используется в качестве признака неопределенности. Величины 00000000 и 00000001 играют особую роль в логических командах — представляют значения логических переменных «истинно» и «ложно». Основным назначением этого типа данных является представление в таких языках, как Паскаль и Ада, так называемых *нумерующих переменных* (задаваемых перечислением их значений).

Ячейка «десятичное число с фиксированной точкой» используется для представления десятичных чисел фиксированного формата с точкой в указываемой позиции. Поле «размер» тега определяет количество цифр в числе. Поле «размер дробной части числа» задает для данного десятичного числа количество

Целое
число



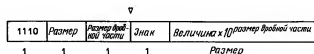
Целое число
увеличенной
разрядности



Порядковое
число



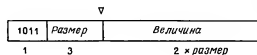
Десятичное
число с фик-
сированной
точкой



Десятичное
число с пла-
вающей
точкой



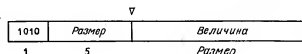
Поле
символов



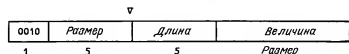
Строка
символов



Поле токенов



Строка
токенов



Указатель



Рис. 14.1. Типы простых ячеек.

цифр, находящихся правее десятичной точки; эта величина может быть меньшей или равной общему количеству цифр числа. Следующие два поля предназначены для знака и величины числа. В поле «знак» размещается знак числа. Величина 0000 в этом поле символизирует положительный знак, —0001 — отрицательный знак числа, 1111 — признак неопределенности. В последнем поле ячейки содержится абсолютное значение числа, умноженное на 10 в степени, равной количеству дробных разрядов. Это число представлено в коде BCD (Binary-Coded Decimal — двоично-кодированный десятичный).

В качестве примера отметим, что переменная с атрибутами FIXED DECIMAL (5,2), имеющая значение 7,9, в ячейке памяти представляется в виде: E52000790. Если бы ее значение было неопределенным, в ячейке было бы записано число E52FXXXXX, где X символизирует неопределенное значение.

Ячейка «десятичное число с плавающей точкой» служит для представления числа в формате мантисса — порядок. Второе поле ячейки (см. рис. 14.1) служит для задания длины мантиссы. Третье поле, как и в ячейке предыдущего типа, несет информацию о знаке. Четвертое поле используется для знака порядка. В пятом поле содержится абсолютная величина порядка в виде десятичного числа, значение которого не выходит за пределы диапазона 0—99. Последнее поле служит для записи десятичной мантиссы. Операции над десятичными числами с плавающей точкой всегда сопровождаются нормализацией мантиссы (путем ее сдвига с целью уничтожения ведущих нулей, если само число не является нулем). Порядок и мантисса представлены в коде BCD.

Ячейки «десятичное число с фиксированной точкой» и «десятичное число с плавающей точкой» позволяют представлять нуль в двух вариантах: +0 и —0. Однако в рассматриваемой системе правильным представлением нуля считается только +0; —0 интерпретируется системой как данные неизвестного типа.

Ячейка «поле символов» используется для записи группы символов фиксированной длины. Во втором поле этой ячейки задается число элементов в группе (от 1 до 4094 символов). Длина третьего поля (измеряемая в токенах) равна удвоенному значению содержимого второго поля. В третьем поле размещаются элементы текущего содержимого ячейки (символы). Содержимое ячейки, равное 11111111, используется в качестве признака неопределенности.

Ячейка «поле токенов» предназначена для хранения элементов, длина каждого из которых равна 4 бит. Во втором поле ячейки указывается количество токенов (эта величина не должна выходить за пределы диапазона значений от 1 до 1 048 574).

Содержимое второго поля определяет длину третьего поля, в котором располагаются хранимые элементы (по одному на токен). Это единственный тип ячейки, для которой неопределенного значения не существует.

Ячейка «строка символов» по структуре похожа на ячейку «поле символов», но может иметь переменную длину. Второе поле в теге этой ячейки предназначено для задания максимально возможного количества символов в строке (1—4094). В третьем поле указывается текущее количество символов в строке (0—4094), т. е. длина строки может динамически изменяться. В четвертом поле размещается сама строка символов. Значение FFF в поле длины является признаком неопределенности для всей строки символов.

Подобным же образом определяется ячейка еще одного типа — «строка токенов». Если в команде специальные указания отсутствуют, ячейка «строка символов» или «строка токенов» используется так же, как и ячейки «поле символов» и «поле токенов». При этом длина строки служит указателем размера поля символов.

Последний тип ячеек для размещения простых данных — ячейка «указатель». В ней может храниться потенциальный адрес объекта (модуля, памяти данных, процесс-машины или порта) или элемента, принадлежащего указанному объекту (ячейки или ее части в модуле, записи активации, памяти данных или точки входа в модуль). Потенциальный адрес состоит из логического адреса и кода доступа. Логические адреса всегда присваиваются машиной и не могут быть изменены программой. Тем не менее программа может копировать содержимое одной ячейки «указатель» в другую ячейку «указатель».

Код доступа в потенциальном адресе содержит информацию о правах, которыми располагает владелец потенциального адреса на доступ к адресуемому элементу. В архитектуре системы SWARD не уточняется положение кода доступа в потенциальном адресе, однако можно отметить, что он состоит из четырех двоичных разрядов следующего значения:

Номер разряда	Вид доступа при нулевом значении разряда	Вид доступа при единичном значении разряда
1	Чтение	Чтение запрещено
2	Запись	Запись запрещена
3	Уничтожение	Уничтожение запрещено
4	Копирование	Копирование запрещено

Значение 1111 имеет смысл признака неопределенности, т. е. указатель содержит неопределенную величину. Вид доступа,

называемый «Копирование», означает возможность копировать потенциальный адрес. Если в потенциальном адресе некоторого указателя не задана возможность копирования, эту ячейку «указатель» нельзя использовать в качестве того операнда команд MOVE или SEND, которым задается пересылаемая величина. Отметим, что точное значение видов доступа «Чтение» или «Запись» зависит от типа адресата, на который ссылается указатель.

Для изменения кода доступа имеется специальная команда, которая, однако, может только понизить степень доступа, ввода на него дополнительные ограничения.

В архитектуре системы SWARD намеренно отсутствует определенная спецификация отдельных битов потенциального адреса. Однако для лучшего понимания архитектуры целесообразно рассмотреть эту спецификацию в конкретной реализации системы SWARD. Под потенциальный адрес отведено 84 бит: 4 бит — для записи кода доступа; 32 бит — для имени объекта, единственного в данной системе; 1 бит — для обозначения, является ли этот потенциальный адрес обычным (непосредственным) или косвенным, и две группы по 23 бит — для обращения, если это необходимо, посредством потенциального адреса к элементу внутри объекта. Если допустить, что новые объекты в системе генерируются в среднем каждые 100 мс (наиболее часто создаваемым объектам — записям активации — потенциальные адреса назначаются только при необходимости), то разнообразия имен, используемых для данной системы, хватит на 13 лет ее эксплуатации. Когда запас имен оказывается исчерпанным, система переходит к их повторному использованию, пропуская имена, которые еще заняты в работе. В результате возможна потенциальная адресация к 2^{32} объектам, каждый из которых может занимать область памяти до 2^{23} токен.

ячейки с вложенными тегами

Имеется пять разновидностей ячеек с вложенными тегами (рис. 14.2). Особенностью этих ячеек является то, что их теги содержат теги ячеек других типов.

Ячейка «массив» служит для представления упорядоченного множества значений данных с одинаковыми атрибутами. Второе поле ячейки этого типа предназначено для хранения двоичного представления общей длины тега; минимальное значение этой величины равно 16. В третьем поле находится размерность массива (1—15), в четвертом — длина (в токенах) текущего содержимого каждого элемента массива. Эта длина может принимать любое значение в диапазоне от 1 до 65 535. Следующие поля ячейки (количество которых равно размерности массива)

каждое длиной 6 токен определяют верхние границы количества элементов массива по соответствующим компонентам его размерности. Результат умножения содержимого всех этих полей на содержимое четвертого поля равен общему количеству

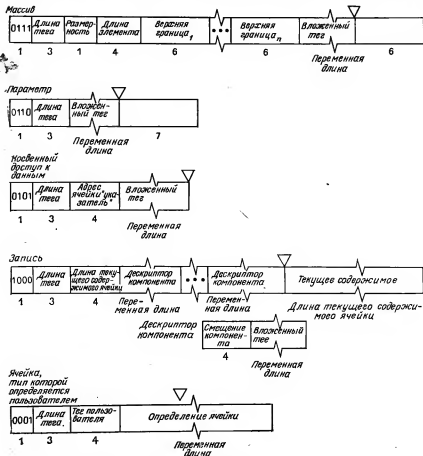


Рис. 14.2. Типы ячеек с вложенными тегами.

токенов, занимаемых элементами массива. По умолчанию предполагается, что нижняя граница индекса по любому компоненту размерности массива равна 1.

Предпоследнее поле ячейки предназначено для вложенного тега, описывающего элемент массива. Длина этого поля равна длине соответствующего вложенного тега. Для элементов массива допустимо использование ячеек следующих типов: ячейки

всех типов для простых данных, ячейки «запись», а также ячейки для простых данных и ячейки «запись», тип которых определяется пользователем. Если в качестве элементов массива используются записи стандартного типа или записи, тип которых определяется пользователем, то компонентами таких записей не могут быть массивы, т. е. архитектура не позволяет к использованию массивы массивов.

Согласно общим принципам рассмотренной структуры ячейки «массив», ее последнее поле, предназначенное обычно для текущего содержимого ячейки, должно рассматриваться как местоположение элементов массива. Однако в архитектуре системы SWARD назначение этого поля специально не определено. Это сделано для того, чтобы предоставить машине, выделяющей область памяти для элементов создаваемого массива, возможность поместить в это поле адрес указанной области, значение которого определяется конкретной реализацией системы.

При работе с массивами формирование необходимых индексов для элементов выполняется системой автоматически; операндами многих команд могут быть как элементы массивов, так и массивы. Приведем пример расположения в ячейке «массив» одномерного массива из 12 элементов, каждый из которых представляет собой строку из 10 символов:

70131000D00000C300AXXXXXX¹⁾

Отметим, что два поля в этой ячейке несут избыточную информацию по отношению к другим данным ячейки. Содержимым этих полей являются длина тега и длина элемента массива. Обе эти величины можно определить, пользуясь содержимым других полей тега (например, информацией из вложенного тега). Эти поля были включены как дополнительные в процессе разработки архитектуры. Они позволяют упростить схемную реализацию и повысить скорость выполнения операций.

Текущее содержимое и, возможно, даже атрибуты ячейки «параметр» определяются динамически во время вызова подпрограммы. Третье поле этой ячейки предназначено для вложенного тега (рис. 14.2). Он задает атрибуты параметра и используется машиной для проверки соответствия между фактическими и формальными параметрами. Допустимыми являются теги ячеек всех типов для простых данных, массива, записи, а также теги ячеек, тип которых определяется пользователем. Длина поля вложенного тега равна длине самого вложенного тега.

¹⁾ Значение четвертого элемента ячейки в этом примере приведено в форме 000D, не учитывающей упоминавшегося выше правила расчета числа элементов в содержательной части ячейки типа «строка символов»; с этим правилом согласуется значение 0017=23₁₆. — *Прим. перев.*

Если вложенный тег в ячейке «параметр» задан нулями, записанными в области размером 6 токен, или является тегом массива, вложенный тег которого представлен нулями, занимающими такую же область, то содержимое ячейки «параметр» называют параметром с динамически определяемым типом. Параметр такого типа динамически принимает атрибуты передаваемого фактического параметра. Если ячейка «параметр» содержит вложенный тег с полем «размер» (например, в ячейках, предназначенных для десятичного числа с плавающей или фиксированной точкой, поля или строки символов или токенов) и в нем содержится нулевое значение или если нулевым является значение в поле «размер» тега массива, вложенного в тег параметра, то содержимое ячейки «параметр» называют параметром с динамически определяемым размером. Содержимое поля «размер» параметра такого типа динамически устанавливается равным значению передаваемого фактического параметра. Если тег, вложенный в тег параметра, является тегом массива, в одном или нескольких подполях верхних границ которого находятся нули, содержимое ячейки «параметр» называют параметром с динамически определяемыми границами. Параметру такого типа по соответствующим компонентам размерности динамически назначаются верхние границы передаваемого массива. Эти особенности передачи параметров рассматриваются в следующем разделе данной главы.

Спецификация последнего поля в ячейке «параметр», так же как и последнего поля в ячейке «массив», данной архитектурой не предусмотрена. Это поле используется машиной SWARD в соответствии с ее конкретной реализацией для отыскания значений параметра (соответствующего фактическому параметру) и для представления неопределенного значения. Независимо от того что подлежит размещению в последнем поле, первоначально машина всегда записывает туда так называемые неопределенные значения, т. е. признак того, что значение содержимого данного подполя еще не определено.

Команды обращаются к ячейке «параметр» так, как если бы она была описана вложенным тегом. Разница заключается лишь в том, что при обращении к параметру машина определяет его значение не непосредственно, а косвенным образом, пользуясь информацией в последнем поле этой ячейки.

Уточним смысл некоторых терминов, и в частности данных и ячеек. Когда речь идет о данных, например, таких типов, как целое число, поле символов, массив указателей целых чисел, то подразумевается ячейка одноименного типа, при обращении к которой можно получить указанные данные. Например, использование термина «целое число» предполагает обращение к ячейке любого типа, текущее содержимое которой имеет целочис-

ленное значение, т. е. к ячейкам «целое число», «параметр целочисленного типа» (параметр с вложенным тегом целочисленного типа), элементу ячейки «массив целочисленного типа», к ячейке «косвенный доступ к данным целочисленного типа», ячейке целочисленных данных, тип которых определяется пользователем, к целочисленному компоненту ячейки «запись», и т. д. Термин «ячейка» употребляется для указания типа области памяти, предоставленной для размещения данных (текущего содержимого ячейки). Так появляются понятия ячейка «целое число», ячейка «параметр» и т. п. Отсюда становится ясна разница между такими понятиями, как целое число и ячейка «целое число».

В рассматриваемой архитектуре предполагается использование ячейки «косвенный доступ к данным», содержащей данные, к которым можно обращаться косвенно, т. е. посредством потенциального адреса. Такая ячейка используется, например, для размещения «базированной переменной» языка ПЛ/1. Для «переменной доступа» языка Ада необходимы ячейка «косвенный доступ к данным» и соответствующая ячейка «указатель».

В третьем поле ячейки «косвенный доступ к данным» расположен адрес ячейки «указатель», используемый для определения местоположения данных, доступ к которым задается содержимым ячейки «косвенный доступ к данным». Адрес, о котором идет речь, является локальным адресом ячейки в пределах заданного адресного пространства; в дальнейшем будет дано более полное определение и объяснение этого понятия. Пока лишь отметим, что данный адрес, задаваемый в ячейке «косвенный доступ к данным», может использоваться для обращения только к ячейкам «указатель» или «указатель параметра».

В последнем поле ячейки «косвенный доступ к данным» размещается вложенный тег, определяющий атрибуты этой ячейки. Допустимы теги ячеек всех типов простых данных, ячеек «запись», «массив», а также ячеек, тип которых определяется пользователем. При выполнении команды, операндом которой является адрес ячейки «косвенный доступ к данным», регистрируется ошибка, если вложенный тег указанной ячейки не совпадает с тегом ячейки, доступ к которой осуществляется косвенным образом посредством указателя. Подобно ячейке «параметр», ячейка «косвенный доступ к данным» может быть ячейкой, тип, размер и границы числа элементов содержимого которой определяются динамически.

Как и при работе с ячейкой «параметр», программа оперирует ячейкой «косвенный доступ к данным» таким образом, как будто бы доступ к данным является не косвенным, а прямым, т. е. как если бы тегом данной ячейки являлся ее вложенный

тег. Машина использует ячейку «указатель» для определения местоположения данных в памяти.

Из описания команды **ACTIVATE**, приведенного в гл. 15 и в разделе данной главы, посвященном ячейкам с динамически определяемыми типом, размерами или границами, следует, что система проверяет соответствие между атрибутами ячейки, к которой установлен косвенный доступ, и атрибутами ячейки «косвенный доступ к данным» (т. е. информацией в ее вложенном теге), подобно тому как проверяется соответствие между атрибутами формальных и фактических параметров при передаче последних. Если соответствие нарушено, регистрируется ошибка «несовместимость операндов».

Ячейка «запись» представляет собой совокупность ячеек, атрибуты которых могут отличаться друг от друга. Эти ячейки содержат не разрозненные данные, а компоненты одной записи. Теги всех отдельных компонентов содержатся в теге ячейки «запись», а текущее содержимое ячеек (значения компонентов) образует текущее содержимое ячейки «запись».

В теге ячейки «запись» содержатся длина тега, общая длина текущего содержимого ячейки и один или несколько дескрипторов компонентов. Дескриптор каждого компонента состоит из смещения длиной 4 токена и тега переменной длины, характеризующего соответствующий компонент. Смещение показывает начало соответствующего компонента при отсчете от первого токена текущего содержимого ячейки «запись». Вложенным тегом в данном случае может быть тег ячейки простых данных любого типа, ячеек «массив», «запись», а также ячеек, тип которых определяется пользователем.

В качестве примера можно указать, что переменная **CHIEF**, определяемая как

```
type PERSON is
  record
    SALARY : delta 0.01 range 0..99999.9;
    SEX    : STRING (1..1);
    AGE    : INTEGER;
  end record;
CHIEF : PERSON;
```

может быть представлена следующей ячейкой «запись»:

```
801C00100000E720008B001000AFFXXXXXXXFF800000
```

Предполагается, что все значения текущего содержимого не определены. Здесь идентификатор **PERSON** задан как ячейка

«запись», а идентификаторы SALARY, SEX и AGE определены не как отдельные ячейки, а как компоненты этой записи. Последовательно расшифровывая тег этой ячейки «запись», получаем следующую информацию. Длина тега ячейки равна 28 токен, а длина ее текущего содержимого — 16 токен. (Числовые значения в тегах представлены в двоичной системе счисления.) Текущее содержимое первого компонента, тип которого E72, имеет нулевое смещение. Текущее содержимое следующего компонента типа B001 располагается со смещением 8, а текущее содержимое компонента типа F (целое число) располагается со смещением 10.

Смещение текущего содержимого любого компонента дескриптора должно равняться сумме смещения текущего содержимого предшествующего компонента и длины текущего содержимого предшествующего компонента, которая может быть извлечена из его тега. Следовательно, текущее содержимое одного компонента даже частично не может перекрываться с текущим содержимым другого компонента. Хотя записи могут включать компоненты в виде записей, в свою очередь включающих компоненты, смещение в дескрипторах любого компонента отсчитывается от начала текущего содержимого записи, внешней по отношению ко всем остальным записям. Возможно, такое решение не является изящным, но оно способствует эффективной реализации системы.

Ячейкой, тип которой определяет или, выражаясь точнее, доопределяет пользователь, может быть ячейка «массив» или ячейка «запись» с префиксом (приставкой) в виде атрибутов, задаваемых пользователем, например, на этапе компилирования. Машина оперирует такой ячейкой, как ячейкой, включенной в описание ячейки другого типа. Разница между непосредственным обращением к некоторой ячейке и обращением к ячейке согласно типу, определяемому пользователем, заключается в том, что в последнем случае в ряде ситуаций машина может использовать для проверки типа информацию, включенную в качестве префикса. В поле «тег пользователя» может быть помещена произвольная информация, представляющая определенные пользователем атрибуты. Например, ячейка с данными 10090001F000008 является ячейкой «целое число», текущее содержимое которой равно +8, а тег пользователя равен 0001.

При использовании ячейки, тип которой определяется пользователем, в качестве операнда команды машина в большинстве случаев игнорирует префикс и оперирует этой ячейкой как ее вложенной частью. Специфика ячейки, тип которой определяется пользователем, проявляется в следующих случаях: когда тег пользователя оказывается вложенным тегом ячейки «параметр»; когда ячейка, тип которой определяется пользователем,

передается в порт или принимается из него; если операндом команды является ячейка «косвенный доступ к данным», тип которой определяется пользователем. Во всех указанных случаях машина выполняет проверку типа с использованием информации в префиксе, добавляемом пользователем.

Итак, если учесть разнообразие возможного задания вложенных тегов, уточнение типа ячейки посредством префикса, добавляемого к ячейке пользователем, атрибуты динамически определяемых типов, размеров и границ, то общее число разнообразных определений ячеек оказывается весьма большим. Примеры обращения к ячейкам многих перечисленных типов приведены в последующих разделах данной главы.

Выше рассмотрено 15 типов ячеек из возможного общего количества типов, равного 16. На основании этого можно было ошибочно заключить, что при необходимости расширения средств архитектуры допустимо введение еще только одного дополнительного типа. В действительности же код 0000 в первых четырех двоичных разрядах ячейки используется как признак расширения типа, в соответствии с которым тип ячейки идентифицируется следующими четырьмя двоичными разрядами. Это позволяет иметь в системе практически неограниченное число различных типов ячеек. Ниже рассматриваются особенности архитектуры, предоставляющие возможность оперировать расширенным набором команд. Используя признаки расширения типа, команды из расширенного набора могут формировать ячейки новых типов. Например, в условиях ориентированного на ФОРТРАН расширенного набора команд ячейка, начинающаяся кодом 00001111, может обозначать комплексное число в языке ФОРТРАН (число с вещественной и мнимой частями). Аналогичного результата можно добиться, снабжая ячейку «записью» с двухкомпонентным текущим содержимым, префиксом (тегом пользователя), т. е. доопределяя тип этой ячейки.

Ниже рассматриваются еще несколько типов ячеек для данных, которые включены в архитектуру в дополнение к описанным выше 15 типам ячеек.

КОСВЕННЫЕ ПОТЕНЦИАЛЬНЫЕ АДРЕСА

Косвенная адресация в рассматриваемой архитектуре основана на применении косвенных потенциальных адресов. Формально косвенный потенциальный адрес (ссылается на другой потенциальный адрес, не являющийся косвенным. Машиной это воспринимается как указание на элемент памяти, адресуемый вторым из этих потенциальных адресов. Любая ссылка средствами косвенной потенциальной адресации через некоторый потенциальный адрес к указываемому им элементу памяти эквивалентна

непосредственной ссылке к элементу памяти с помощью этого потенциального адреса. Разница состоит лишь в том, что при использовании косвенной адресации применяется код доступа из косвенного потенциального адреса. Любая операция (пересылка, сравнение и т. п.) с указателем, содержащим косвенный потенциальный адрес, приводит к тому же результату, что и операция с указателем, содержащим обычный (некосвенный) потенциальный адрес. Например, если указатель А содержит косвенный потенциальный адрес другого указателя В, любое использование в командах указателя А для обращения к памяти эквивалентно использованию указателя В, за исключением того, что в операции участвует код доступа указателя А, а не В. Любые операции, выполняемые непосредственно с указателем А, ссылаются только на него, а не на указатель В. Косвенная адресация используется при выполнении так называемого разрешения потенциальных адресных ссылок, например при обращении к ячейке «косвенный доступ к данным» и выполнении команды CALL или команды SEND, пересылающей данные в порт.

Косвенную потенциальную адресацию можно использовать в различных целях. В частности, с ее помощью можно управлять санкционированным доступом. Например, было бы желательно, чтобы программа А предоставила программе В доступ к некоторым данным, сохранив за собой возможность лишить программу В этого права на доступ в любой момент времени. Это достигается предоставлением программе В косвенного потенциального адреса указателя, содержащего непосредственный (прямой) потенциальный адрес указанных данных. Программа А может в любой момент изменить содержимое указателя и тем самым лишить программу В доступа к этим данным. Косвенную адресацию можно использовать также для динамической смены объектов или модулей без специального переопределения связей этих объектов или программ. В частности, если модуль Х вызывает модуль У посредством косвенного потенциального адреса, можно выполнить замену У на новый модуль изменением того (прямого) потенциального адреса, который непосредственно указывает на вызываемый модуль. При этом не требуется никаких изменений в модуле Х. Еще одна возможность применения косвенных адресов связана с организацией управления доступом к объектам, что характерно, например, для операционных систем. В частности, функцию операционной системы, состоящую в управлении использованием объектов как ресурсов с возможностями запроса монопольного или совместного управления, лучше всего организовать на основе обращения программ к косвенным, а не непосредственным потенциальным адресам.

ячейки с динамически определяемыми типом, размером или границами

Выше отмечалось, что ячейки «параметр» и «косвенный доступ к данным» допускают динамическое задание типа, размера и границ их текущего содержимого. Эти свойства позволяют формировать программы, в значительной степени не зависящие от атрибутов обрабатываемых ими данных.

Ячейки «параметр» с вложенными ячейками «поле символов», «строка символов», «поле токенов», «строка токенов», «десятичное число с фиксированной точкой» или «десятичное число с плавающей точкой» допускают динамическое изменение размера их текущего содержимого, если во вложенном теге ячейки «параметр» в поле «размер» записать нули. Например, код 6008B003XXXXXXX представляет ячейку «параметр» с вложенной ячейкой «поле символов», размер текущего содержимого которой равен 3; что же касается кода 6008B000XXXXXXX, то он представляет подобную ячейку «параметр», допускающую динамическое задание указанного размера. Аналогичным образом ячейки «параметр» с вложенными ячейками «массив» (которые в свою очередь являются совокупностью полей и строк символов, десятичных чисел с фиксированной и плавающей точками) становятся ячейками с динамически изменяемым размером текущего содержимого, если присвоить нулевое значение содержимому вложенного тега массива.

Описываемый механизм динамического определения характеристик данных схож с использованием в языке ПЛ/1 для подобных целей символа «звезда», но более удобен для разработки программ. Приведем примеры параметров с динамически определяемым размером и соответствующие решения средства языка ПЛ/1:

	Q: PROCEDURE(A,B);
6007E0XXXXXXXX	DECLARE FIXED DECIMAL(*);
601770131XXXX000009B000XXXXXX	DECLARE B(9) CHARACTER
	(*);

Поле длины элемента во вложенном теге массива в ячейке «параметр» или «косвенный доступ к данным» не используется и может заполняться любым содержимым. Как обычно, символ X обозначает произвольное значение.

Если формальный параметр процедуры наделен свойством динамического определения размера, он автоматически получает значение атрибута «размер» соответствующего фактического параметра процедуры. Ниже в определении команды ACTIVATE перечислены правила соответствия типов формальных и фактических параметров.

Динамическое задание размеров текущего содержимого возможно для полей и строк символов, целых чисел, чисел с фиксированной или плавающей точкой, а также для массивов, если доступ к ним осуществляется косвенно, т. е. посредством ячейки «косвенный доступ к данным». Динамическое задание размеров для ячейки «косвенный доступ к данным» осуществляется подобно тому, как это делается для ячейки «параметр». Обратимся к следующим примерам:

```
500CYYYY3000      DECLARE A CHAR(*) VARYING
                    BASED (P);
501AYYYY70121XXXX000009E0X  DECLARE B(9) FIXED DECIMAL
                              (*) BASED (P);
```

В этих примерах YYYYY обозначает адрес ячейки указателя Р. Запись на языке ПЛ/1 носит условный характер, так как правила этого языка не позволяют откладывать взаимную привязку переменных на более поздний этап формирования рабочей программы.

При каждом обращении к ячейке «косвенный доступ к данным», используемой в режиме динамического определения размера текущего содержимого ячейки косвенно адресуемых данных, первая из упомянутых ячеек получает значение размера содержимого второй ячейки. Согласование между этими параметрами двух ячеек аналогично тому, которое осуществляется между формальными и фактическими параметрами процедуры.

Динамическое определение размеров того или иного текущего содержимого не может быть использовано применительно к ячейке «запись», но допускается при работе с ячейкой «параметр» или «косвенный доступ к данным», тип которой определяется пользователем.

Режим динамического определения границ числа элементов массива задается записью нулей в одном или нескольких полях верхних границ вложенного тега этого массива. Приведем примеры:

```
601C70182XXXX000000000002E42XXXXXX  Q: PROCEDURE(A,B);
                                           DECLARE   A(*,2)
                                           FIXED DEC (4,2);
601770131XXXX000000B000XXXXXX         DECLARE   B(*)
                                           CHAR(*);
```

Как следует из второго примера, режимы динамического определения границ и размера реализуются независимо друг от друга, т. е. массив параметров может обладать обоими этими

возможностями. Если последний используется в режиме динамического определения границ, то для каждого компонента размерности, имеющего нулевое содержимое поля верхней границы, автоматически назначаются соответствующие верхние границы массива фактических параметров.

Любая ячейка «косвенный доступ к данным» может использоваться также и в режиме динамического определения границ косвенно адресуемого массива. Это достигается способом, аналогичным описанному выше.

```
501BYYYY70131XXXX000000B000    DECLARE A(*) CHAR(*)
                                   BASED (P);
```

В данном примере, как и в предыдущем, ячейка обладает свойствами динамического определения как размера, так и границ.

При каждом обращении к ячейке «косвенный доступ к данным», используемой в режиме динамического определения границ косвенно адресуемого массива, она получает значения границ этого массива для всех компонентов, представленных в ней нулевыми значениями. Режим динамического определения границ компонентов массива может использоваться также для массивов внутри записей и массивов, тип которых задается пользователем.

Для использования ячейки «параметр» в режиме динамического определения типа данных 6 токен вложенного тега этой ячейки должны иметь нулевые значения. Для задания такого же режима для параметра в виде массива указанное количество нулей должно быть помещено во вложенный тег (описывающий элемент массива) внутри вложенного тега массива.

```
600A000000XXXXXXX                Q: PROCEDURE (A,B);
601970151XXXX000000000000XXXXXX    DECLARE A D-TYPED;
                                   DECLARE B(*) D-TYPED;
```

В первом из приведенных здесь примеров в режиме динамического определения типа в качестве параметра используется скалярная величина. В этом случае он автоматически получает все атрибуты соответствующего фактического параметра. Последний, однако, не может быть записью, массивом или элементом, тип которого определяется пользователем. Если в режиме динамического определения типа в качестве параметра используется массив, он автоматически получает все атрибуты элементов соответствующего массива фактических параметров.

Второй пример показывает задание режима динамического определения не только типа элементов массива, но и границ их компонентов.

Режим динамического определения типа для ячейки «косвенный доступ к данным» может быть задан помещением нулей в область из 6 токенов вложенного тега. С целью задания такого же режима для подобной ячейки, обеспечивающей доступ к массиву, указанное количество нулей необходимо поместить во вложенный тег вложенного тега массива. Подобный режим может быть задан для любой ячейки «косвенный доступ к данным», адресуемой к массиву:

500EYYYY000000	DECLARE A D-TYPED BASED (P);
501DYYYY70151XXXX00000000000000	DECLARE B(*) D-TYPED BASED (P);

Во втором примере для ячейки «косвенный доступ к данным» заданы одновременно как режим динамического определения типа, так и режим динамического определения границ.

Если в режиме динамического определения типа данных используется ячейка «косвенный доступ к данным», адресуемая к данным скалярного типа, как в первом примере, то при каждом обращении к ячейке с этими данными ячейка «косвенный доступ к данным» получает все атрибуты косвенно адресуемой ячейки. В качестве последней нельзя использовать ячейки «запись», «массив», а также ячейки, тип которых определяется пользователем. Если ячейка «косвенный доступ к данным» и используется в режиме динамического определения типа данных, представляющих собой массив, то при каждом обращении к ней она получает все атрибуты элементов такого косвенно адресуемого массива.

Неправильно было бы считать, что возможности динамического определения типов и размеров данных, а также границ массивов делают архитектуру SWARD менее надежной и защищенной от несанкционированного доступа. Использование этих возможностей в сочетании с теговой памятью и системой команд, инвариантных к типу обрабатываемых данных, позволяет создавать программы, в значительной степени независимые от обрабатываемых данных. При несовпадении типов данных (например, при попытках выполнить арифметические операции над полями символов) ошибка будет обнаружена и при использовании режима динамического определения соответствующих характеристик ячеек. Возможно только, что она будет обнаружена несколько позднее, чем если бы этот режим не использо-

вался. Например, если формальный параметр определен как одномерный массив из 10 элементов типа «поле символов», размер каждого из которых равен 6, то при вызове процедуры или модуля система обнаружит ошибку, если соответствующий фактический параметр не обладает такими же характеристиками. Однако, если формальный параметр используется в режиме динамического определения типа и границ, проверка формального параметра будет выполняться только на одномерность массива, передаваемую в качестве фактического параметра. Если при последующем выполнении процедуры встретится команда с операндами, предполагающими другие характеристики массива (например, с попытками обращения к несуществующему элементу, обращения с элементами как с числами, когда в действительности они таковыми не являются, или обращения к несуществующим частям элемента массива, являющегося полем или строкой), то регистрируется ошибка.

ОБЪЕКТЫ СИСТЕМЫ

В системе различаются объекты пяти типов: модули, записи активации, память данных, процесс-машины и порты.

МОДУЛЬ

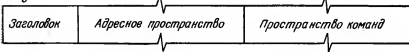
Главным объектом в процесс-машинах является модуль. Модуль состоит из последовательности машинных команд и адресного пространства данных, которыми оперируют машинные команды.

Модуль в системе SWARD соответствует таким понятиям в языках программирования, как пакет в языке Ада, внешняя процедура и функция в языке ПЛ/1, подпрограмма в языке КОБОЛ и подпрограмма SUBROUTINE в языке ФОРТРАН. Модуль создается с помощью команды CREATE-MODULE. Эта команда обращается к *внешней форме модуля* (рис. 14.3), представленной в виде строки токенов, и на ее основе создает объект «модуль». Следовательно, в архитектуре SWARD структура модуля не является раз и навсегда определенной: она задается описанием упомянутой выше так называемой *внешней формы модуля* (сокращению *внешний модуль*). Модуль может быть разрушен командой DESTROY или в момент уничтожения процесс-машины; последняя возможность может быть запрошена при необходимости.

Внешний модуль выступает в качестве основного интерфейса с системой, поскольку представляет собой выходные данные компилятора. Согласно рис. 14.3, внешний модуль состоит из трех частей: заголовка фиксированной длины и имеющих переменную длину адресного пространства и пространства команд.

В заголовке модуля указываются некоторые атрибуты модуля и определяются составные элементы других частей модуля. Каждое из первых четырех полей заголовка состоит из 5 токенов, содержащих внутренние указатели на начало отдельных основных областей в модуле. Кроме того, эти указатели используются системой для расчета последних информационных разрядов предыдущих областей. Поэтому упомянутые основные

Модуль



Заголовок

Указатель динамической части адресного пространства	Указатель статической части адресного пространства	Указатель области команд	Указатель конца модуля +1	CAS	IAS	SIS	Коды обрабатываемых ошибок
---	--	--------------------------	---------------------------	-----	-----	-----	----------------------------

Адресное пространство

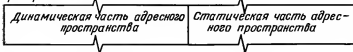


Рис. 14.3. Структура внешнего модуля.

области должны быть смежными. Если некоторая область отсутствует, предназначенный ей указатель устанавливается на начало следующей области. Например, если отсутствует динамическая часть адресного пространства, указатель этой области и указатель статической части адресного пространства имеют одно и то же значение.

Два поля, CAS и IAS, длиной 1 токен используются для задания длины адресов ячеек и команд в командах данного модуля. Каждое поле может содержать двоичную величину, указывающую количество токенов в адресах этого модуля. Понятия «адрес ячейки» и «адрес команды» поясняются ниже в разделе, описывающем форматы команд. (Отметим, что адреса ячеек, содержащиеся в ячейках «косвенный доступ к данным», имеют фиксированную длину, равную 4 токенов.)

Поскольку адресное пространство модуля ограничено только теми ячейками, которые определены в этом модуле, целесообразно ограничивать длину используемых адресов наименьшим достаточным значением. Так, вместо того чтобы в командах задавать для адресов поля фиксированной длины, оказывается более предпочтительным подбирать размеры этих полей для

каждого модуля. Необходимо только, чтобы длина адресов для обращения к ячейкам была достаточной для адресации ко всем имеющимся ячейкам в модуле, т. е. для адресного пространства модуля. Аналогичным образом необходимо, чтобы длина адресов для обращения к командам была достаточной для адресации ко всем имеющимся командам области команд модуля. В соответствии с этим при работе с модулями, в которых имеется мало ячеек небольшого размера, можно ограничиться короткими адресами ячеек; при работе же с модулями более крупных размеров, содержащими большее число ячеек, потребуются более длинные адреса.

Архитектура системы SWARD принципиально допускает выбор значений CAS и IAS. Однако в реализованном варианте системы значения CAS и IAS приняты постоянными и равными трем.

Следующее поле заголовка модуля SIS — идентификатор конкретного расширенного множества команд — состоит из 1 токена и определяет исходный язык программирования модуля. Включение этого поля в состав заголовка модуля объясняется стремлением расширить базовое множество команд дополнительными командами, отражающими специфику языков программирования. Например, если в поле SIS находится 0, код операции 0087 может оказаться недопустимым. Если значение содержимого SIS равно 1, код 0087 может соответствовать команде PICTURE — редактирование данных с шаблоном — при использовании языка ПЛ/1. Если содержимое указанного поля равно 2, тот же код 0087 может обозначать одну из команд обращения к базе данных. Приведенный пример указывает и на другое достоинство системы: оказывается возможным освободить разработчиков отдельных систем (например, разработчиков транслятора КОБОЛа) от необходимости принимать во внимание информацию о командах, связанных с операционной системой. Команды, не входящие в определенные множества программ и не используемые отдельными программистами, оказываются как бы скрытыми от них, что представляет собой определенное удобство.

Наличие подобного поля расширения набора команд или средств языка программирования позволяет динамически изменять часть набора команд системы, предоставляя системным программистам возможность видоизменять набор команд в соответствии с меняющимися требованиями таким образом, чтобы это оставалось «незамеченным» для уже разработанных и действующих программ. В существующем варианте реализованной системы SWARD возможность включения дополнительных множеств команд не используется.

Последнее поле заголовка модуля имеет длину 6 токенов и содержит коды ошибок (обнаруживаемых во время работы ма-

шины), которые подлежат обработке по заявке данного модуля. Назначение этого поля рассматривается в одном из следующих разделов, посвященном обработке ошибок.

Второй частью внешнего модуля является его адресное пространство. Оно содержит группы ячеек, определяющих данные, к которым может обращаться данный модуль. Порядковый номер первого токена любой ячейки адресного пространства определяется как адрес ячейки. Так, адрес первой ячейки равен 1, адрес второй ячейки на единицу больше длины первой ячейки и т. д.

Хотя программа воспринимает адресное пространство как единое целое, в нем можно выделить две части. Они используются системой для распределения областей памяти и управления доступом к ним.

В *динамической части адресного пространства* расположены ячейки, играющие роль динамически распределяемой памяти при вызове модуля. Когда такой вызов осуществляется, система формирует запись активации и копирует в нее динамическую часть адресного пространства. Если команда программы обращается к ячейке динамической памяти адресного пространства, система автоматически преобразует адрес этой ячейки в адрес ее копии в записи активации модуля.

Машина строго поразрядно копирует динамическую часть адресного пространства в запись активации. В результате компилятор может обеспечить динамической переменной¹⁾ возможность получения начального значения путем записи требуемого значения в ячейку этой переменной в динамической части адресного пространства. Если динамическая или какая-либо другая переменная не имеет начального значения, то компилятор помещает в ячейку признак неопределенности (неопределенное значение). Исключением из этого общего правила являются ячейки «указатель». В целях гарантии защиты от несанкционированного доступа установка начальных значений ячеек «указатель» при создании объекта выполняется помещением в них признака неопределенности. В динамической части адресного пространства могут присутствовать ячейки всех типов. Если в динамической части адресного пространства размещается ячейка «массив», то область памяти для хранения всех элементов массива выделяется в записи активации. Исходные значения этих элементов задаются как неопределенные. Ячейки «параметр» должны располагаться только в этой части адресного пространства. При создании модуля исходные значения этих ячеек также назначаются неопределенными.

¹⁾ Динамическими называются переменные, выделение и освобождение памяти под которые осуществляется системой соответственно при каждом входе и выходе из блока программы. — *Прим. перев.*

Статическая часть адресного пространства содержит ячейки переменных, для которых память выделяется только один раз перед выполнением (т. е. при выполнении команды CREATE-MODULE). Если такой переменной, называемой *статической*, должно быть присвоено исходное значение, его следует поместить в ячейку этой переменной в статической части адресного пространства. В противном случае в ячейке должен находиться признак неопределенности (этот признак устанавливается системой в качестве начального значения для всех ячеек «указатель»). В статической части адресного пространства могут встречаться все типы ячеек, за исключением ячеек «параметр». Начальные значения элементов массивов устанавливаются неопределенными.

Последняя часть внешнего модуля — область команд, содержащая последовательность машинных команд. Большинство машинных команд имеет переменную длину. Порядковый номер первого токена команды в области команд считается адресом команды. Адрес первой команды равен 1, адрес второй команды на единицу больше длины первой команды и т. д.

ЗАПИСЬ АКТИВАЦИИ

Объект «запись активации» содержит пространство для ячеек, расположенных в динамической части адресного пространства модуля. Этот объект создается при активации модуля с помощью команды CALL и уничтожается, когда данный модуль возвращает управление вызывающему модулю или когда уничтожается соответствующая процесс-машина. В программе не предусмотрено средств для обращения к записи активации (адресация к ней выполняется неявно, через динамическую часть адресного пространства), поэтому описание архитектуры системы SWORD не содержит более подробных сведений о записи активации.

ПАМЯТЬ ДАННЫХ

Объект «память данных» создается программой, согласно алгоритму работы которой необходимо динамически выделять память для ячейки «косвенный доступ к данным». Другими словами, выделяется память на случай динамического создания ячейки системой. Эти действия выполняются по команде ALLOCATE. Память закрепляется за модулем временно до тех пор, пока не будет выдана команда DESTROY или в случае, если указан специальный необязательный параметр в команде ALLOCATE—до уничтожения процесс-машины. Так как к объекту «память данных» программа обращается не прямо, а посред-

ством ячейки «косвенный доступ к данным», то в документации рассматриваемой архитектуры отсутствует более подробное описание этого объекта.

ПРОЦЕСС-МАШИНА

Спецификой объекта, именуемого процесс-машиной, является тот факт, что он параллельно с другими процесс-машинами выполняет один или несколько модулей архитектуры системы SWARD. Выполнение команд модулей процесс-машиной называется *процессом*. Процесс-машина создается в результате выполнения другой процесс-машиной команды CREATE-PROCESS-MACHINE. Уничтожается процесс-машина одним из следующих способов: командой DESTROY; по команде возврата в инициализированный процесс; при появлении ошибки такого типа, для которого данный процесс не располагает соответствующими средствами обработки ошибок, или при уничтожении процесс-машины, которая выполняла процесс, породивший данную процесс-машину, если в команде CREATE-PROCESS-MACHINE указан соответствующий необязательный параметр.

Процесс-машину можно представить как состоящую из программ обработки команд (собственно «машины»), стека записей активации, области распределяемой памяти и регистратора состояния. Однако в рамках архитектуры понятие «процесс-машина» детализируется лишь в степени, необходимой для описания команд системы.

ПОРТ

Порт является абстрактным объектом в системе, служащим для установления связей между двумя или большим числом процессов в целях их согласованной совместной работы. Объект «порт» создается командой CREATE-PORT, а разрушается или командой DESTROY, или при уничтожении породившей его процесс-машины, если в команде CREATE-PORT указан соответствующий необязательный параметр. О наличии объекта «порт» в системе можно судить только по семантике (смысловому содержанию) двух команд — SEND и RECEIVE. Никакой дополнительной информации об этом объекте в описании рассматриваемой архитектуры не приводится.

ФОРМАТЫ КОМАНД И СПОСОБЫ АДРЕСАЦИИ

Формат машинной команды состоит из *поля кода операции* и одного или нескольких следующих за ним адресных полей. В некоторых командах имеется одно поле адреса, в других —

фиксированное число этих полей (до 5). Имеются также команды, число адресных полей в которых может меняться.

ПОЛЕ КОДА ОПЕРАЦИИ

Первым полем любой команды является поле ее кода операции. Длина этого поля не является одной и той же для всех команд. Она выбирается в зависимости от относительной частоты использования той или иной команды. Так, в 15 наиболее распространенных командах для этого поля отведен 1 токен, для следующих по частоте использования 15 команд — 2 токена, а для остальных — 4 токена.

АДРЕСНЫЕ ПОЛЯ

В системе различают семь типов адресных полей, объединяемых в три группы полей: адрес операнда, адрес команды и непосредственные данные. *Адрес операнда* предназначен для обращения к операндам в адресном пространстве. Имеется шесть разновидностей адреса операнда.

Адрес ячейки. Адрес ячейки занимает N токен и служит для ссылок на ячейки адресного пространства (здесь N — значение содержимого поля длины адреса ячейки, входящего в заголовок модуля). Например, если значение $N(CAS)$ равно 2, адрес операнда, равный 1A, указывает на ячейку, начинающуюся с 26-го токена в адресном пространстве модуля. Адреса ячеек не могут использоваться для обращения к массивам или записям.

Литерал. Поле, содержащее адрес операнда в форме литерала, состоит из N токенов, в которых находятся нули, и следующего за ними 1 токена, в котором хранится число от 0 до 9. Например, если $N(CAS) = 2$, адрес операнда, равный 004, является литералом со значением +4. При использовании литерала в логических командах он может принимать одно из двух значений: 0 (ложно) и 1 (истинно).

Адрес элемента массива. Адрес элемента массива состоит из $D+1$ полей (предполагается, что массив имеет размерность D). Первое поле служит для хранения адреса ячейки массива или адреса компонента записи для массива (т. е. для массива как компонента записи). В следующих D полях должны располагаться литералы либо адреса ячеек «целое число» или «параметр», если содержимое последних определяется как целочисленное значение. Например, если адрес ячейки A «массив» равен 20_{16} , адрес переменной (адрес ячейки) I равен $3C_{16}$ и $N = 2_{16}$, то адрес операнда для $A(I)$ равен $200043C_{16}$. Если $N = 3_{16}$, то адрес операнда принял бы значение, равное $020000403C_{16}$.

Адрес массива. Адрес массива относится ко всему массиву. Способ его записи подобен записи адреса элемента массива, только во всех полях для индексов отдельных компонентов размерности должен быть записан символ *. Этот символ представляется полем типа литерал со значением, равным F_{16} (1111). Тогда адрес упомянутого выше массива А имеет вид 2000F00F₁₆.

В соответствии с принятыми формами записи адресов имеется принципиальная возможность для обращения к кросс-секциям массивов, например, можно пользоваться выражениями вида $A(*, I)$ языка ПЛ/1. Однако реализация этой возможности в системе SWARD не предусмотрена.

Адрес компонента записи. Этот адрес состоит из двух полей. Первое из них имеет форму адреса ячейки, адреса элемента массива или адреса массива для записи (последние два варианта относятся к случаям, когда запись является элементом массива). Второе поле предназначено для адреса компонента. Если компонент не является массивом, это поле состоит из 3 токенов, в которых находится смещение дескриптора данного компонента от начала тега записи. Если компонент представляет собой массив, то вслед за указанной информацией (3 токенов) следует D_{16} литералов или адресов ячеек «целое число» либо «параметр», если содержимое последней определяется как целочисленное значение (для адресации элемента), или D_{16} значений * (для адресации всего массива). Отметим, что обращение к компонентам, которые сами являются записями, выполняется с помощью адресов компонентов записи, а обращение к компонентам компонентов записей происходит как к компонентам записи, внешней по отношению ко всем другим записям.

Адрес записи. Адрес записи относится полностью ко всей записи. Форма представления адреса этого типа совпадает с формой адреса компонента записи. Но в поле адреса компонента, состоящем, как отмечалось выше, из 3 токенов, находятся нули.

При отсутствии специального указания в качестве адреса операнда в командах можно использовать любую из его шести рассмотренных здесь форм. Не допускается только использование литерала в качестве операнда принимающего поля команды. Под *операндом* понимают данные, к которым обращается команда, использующая адрес операнда (возможно, через ячейку «параметр» или «косвенный доступ к данным»). *Операндом принимающего поля* называют операнд, значением которого является результат операции.

Рассмотренные выше типы содержимого адресных полей образуют одну из трех возможных разновидностей адресов записи. Ко второй разновидности относится *адрес команды*. Этот адрес занимает область памяти М токенов (М — значение содержимого поля длины адреса команды в заголовке модуля) и

позволяет ссылаться на команды, принадлежащие области команд.

К последней разновидности адресов записи относится содержимое поля так называемых *непосредственных данных*. Это поле длиной 1 или 2 токены содержит не адрес, а некоторую величину, которая непосредственно используется в команде. Поскольку эти операнды находят применение в ограниченном числе команд и для специальных целей, более подробно они рассматриваются ниже при описании соответствующих команд.

ПРАВИЛА ФОРМИРОВАНИЯ АДРЕСОВ ОПЕРАНДОВ

С помощью адресов операндов могут выполняться сложные способы адресации, позволяющие, например, с помощью единственного адреса обратиться к компоненту массива записей, являющегося в свою очередь компонентом записи. Подобная гибкая форма адресации требует особого внимания к порядку указания отдельных элементов адреса, поскольку допускается даже рекурсивное образование адресов, при котором, например, адрес элемента массива может содержать адрес компонента записи или наоборот.

Введем сокращенные обозначения для рассмотренных выше типов адресов операндов:

са — адрес ячейки,
lit — литерал,
aea — адрес элемента массива,
aa — адрес массива,
rsa — адрес компонента записи,
ra — адрес записи.

Пусть также $x(a)$, $x(i)$ и $x(r)$ обозначают адрес типа x , используемый для обращения к данным типа массив, целое число и запись соответственно. Тогда

```
aea ::= са(a) ea...|rsa(a) ea...
aa  ::= са(a)*...|rsa(a)*...
rsa ::= rcal pa
ra  ::= rcal 000
```

где

```
pa  ::= cdo|cdo ea...|cdo*...
ea  ::= са(i)|lit
*   ::= 0F"при N=1"|00F "при N=2"|...
rcal ::= са(r)|aea(r)|aa(r)
cdo  ::= "3 токен с ненулевым значением"
```

Примеры структуры адресов операндов, формируемой в соответствии с этими правилами, приведены в табл. 14.1.

Таблица 14.1. Примеры адресов операндов (массивы предполагаются одномерными)

Тип адресуемых данных	Тип ячейки, которой принадлежат данные	Тип адреса операнда	Структура адреса операнда
Данные простого типа	—	ca	ca
Литерал	—	lit	lit
Данные простого типа	Массив	aea	ca ea
Все элементы, являющиеся данными простого типа	Массив	aa	ca*
Запись	—	ga	ca 000
Компонент — не массив	Запись	gca	ca cdo
Компонент — не массив	Массив записей	gca	ca ea cdo
Все компоненты (срез)	Массив записей	gca	ca* cdo
Запись	Массив записей	ga	ca ea 000
Все записи	Массив записей	ga	ca* 000
Элемент (не массив) массива — компонента	Запись	aea	ca cdo ea
Все элементы (не массивы) массива — компонента	Запись	aa	ca cdo*
Компонент (не массив) записи, являющейся элементом массива — компонента	Запись	gca	ca cdo ea cdo
Все компоненты (не массивы) записи, являющейся элементом массива — компонента	Запись	gca	ca cdo* cdo
Запись, являющаяся элементом массива — компонента	Запись	ga	ca cdo* 000

Общие принципы формирования адресов могут быть сформулированы также несколько иначе. Общий адрес разбивается на элементарные компоненты. Первый элементарный компонент адреса должен относиться к ячейке, содержащей адресуемые данные, а последний элементарный компонент адреса должен относиться собственно к адресуемым данным. Иначе говоря, если ссылка устанавливается на какие-либо компоненты ячейки, то каждый последующий элементарный компонент адреса должен относиться к следующему, более глубокому уровню вложения адресуемого компонента.

ОБРАБОТКА ОШИБОК В СИСТЕМЕ SWARD

Одной из основных задач машины рассматриваемой архитектуры является обнаружение определенного класса ошибок, совершаемых при программировании. Поэтому велико значение

применяемых в системе методов обнаружения ошибок и информирования о них пользователя. В этом разделе рассматриваются типы ошибок, которые способна выявлять система, информация об ошибках, передаваемая системой соответствующему процессу, и средства взаимодействия между процессом и системой при обработке ошибок.

ТИПЫ РЕГИСТРИРУЕМЫХ ОШИБОК

Ниже перечисляются типы ошибок, регистрируемых в машине SWORD и разъясняются условия их возникновения. При выполнении команды может быть одновременно зарегистрировано несколько ошибок, однако архитектура не содержит средств указания ни первой из встретившихся ошибок, ни порядка, в котором были обнаружены ошибки отдельных типов.

Ошибка типа 1 — *неверный код операции* — регистрируется при попытке выборки команды с неправильным кодом операции или при выходе за пределы области, отведенной под команды, при попытке адресации команды.

Ошибка типа 2 — *неправильная адресация* — возникает в одной из следующих ситуаций:

1) используемый адрес ячейки выходит за пределы для пространства адресов данного модуля или относится не к той части (статической вместо динамической или наоборот) адресного пространства;

2) индекс элемента массива не является целым числом;

3) ошибочный адрес операнда;

4) внешнее обращение к компонентам модуля (по командам LINK или COMPUTE-ENTRY-CAPABILITY) не соответствует установленным правилам адресации;

5) ошибка при обработке адреса ячейки (например, нарушены правила использования ячейки «косвенный доступ к данным»);

6) наличие замкнутой петли адресации при использовании косвенных потенциальных адресов (например, указатель ссылается сам на себя).

Ошибка типа 3 — *данные неизвестного формата* — регистрируется при обращении к ячейке с нераспознаваемым типом данных или значением.

Ошибка типа 4 — *нарушение защиты от несанкционированного доступа* — имеет место в одном из следующих случаев:

1) обращение к ячейке (для записи, чтения или уничтожения) посредством потенциального адреса с кодом доступа, не соответствующим запрошенной операции;

2) непосредственный запрос на уничтожение части памяти, принадлежащей записи активации или модулю;

3) попытка программы изменить значение переданного модулю фактического параметра, при работе с которым санкционировано только «чтение»;

4) попытка программы копировать содержимое ячейки «указатель» при отсутствии соответствующего санкционирующего признака в коде доступа.

Ошибка типа 5 — *недействительный потенциальный адрес* — возникает при использовании потенциального адреса, содержащего логический адрес, который машине не известен (например, объект, к которому производится ссылка, уничтожен).

Ошибка типа 6 — *выход индекса за допустимые границы* — регистрируется при обращении к элементу массива, когда значение индекса этого элемента выходит за допустимые пределы, или при обращении к элементу строки с порядковым номером, превышающим текущую длину строки.

Ошибка типа 7 — *неверный тип операнда* — имеет место, когда тип операнда не соответствует допустимому для данной команды типу (типам) или при адресации объектов «память данных», недопустимых к использованию данной командой.

Ошибка типа 8 — *неопределенный операнд* — возникает при попытке использовать значение операнда, которое задано как «неопределенное». Этого состояния не бывает при выполнении команды DEFINED-BRANCH-FALSE, которая, по существу, является средством проверки неопределенности значения операнда.

Ошибка типа 9 — *несовместимые операнды* — регистрируется, если два или несколько операндов одной и той же команды взаимно несовместимы. Условия совместимости операндов приводятся в определениях соответствующих команд. Ошибка такого рода может быть, в частности, зарегистрирована при выполнении команды ACTIVATE, если тип ячейки «параметр» несовместим с типом ячейки, содержащей соответствующий фактический параметр, а также в случае выполнения команд RECEIVE и SEND при несоответствии типов операндов для данных при их передаче и приеме. Это же состояние регистрируется, если атрибуты ячейки «косвенный доступ к данным» не согласуются с атрибутами адресуемых данных.

Ошибка типа 10 — *переполнение* — фиксируется, когда операнд, указывающий местоположение результата выполнения команды, является адресом ячейки, размер которой недостаточен для размещения этого результата. Для операндов арифметических операций переполнение наступает, если происходит потеря ненулевых старших разрядов числа. Аналогичная ситуация возникает при выполнении операций над числами с плавающей точкой, когда показатель степени результата оказывается больше чем 99. Если операнды являются строками симво-

лов, то переполнение наступает в тех случаях, когда размер памяти, предусматриваемый командой для хранения результата, недостаточен для строки, формируемой этой командой.

Ошибка типа 11 — *потеря значимости* — регистрируется тогда, когда результат выполнения операции над числами с плавающей точкой является числом, показатель степени которого меньше чем —99.

Ошибка типа 12 — *некорректное деление* — происходит при попытке деления на нуль.

Ошибка типа 13 — *недействительный модуль* — возникает при выполнении команды CREATE-MODULE, если допускается неправильный формат внешнего модуля.

Ошибка типа 14 — *недопустимая передача управления* — обусловлена разнообразными причинами при выполнении команды, передающей управление. Наиболее часто подобная ошибка регистрируется при попытке передачи управления за пределы области команд данного модуля.

Ошибка типа 15 — *неправильное число передаваемых данных* — возможна при выполнении команды ACTIVATE, если количество формальных параметров, задаваемое при описании программы, не равно количеству фактически передаваемых параметров. Кроме того, она возникает при выполнении команд RECEIVE и SEND, когда количество посылаемых параметров (определяемых командой SEND) не соответствует количеству принимаемых параметров (определяемых командой RECEIVE).

Ошибка типа 16 — *недопустимое преобразование данных* — регистрируется при выполнении команды CONVERT, если операнды не удовлетворяют требованиям, предъявляемым к ним правилами согласования, которые определены в описании команды CONVERT.

Ошибка типа 17 — *трассировка* — наблюдается при выполнении команд в модуле, для которого установлен режим трассировки (см. команду TRACE в гл. 15).

Ошибка типа 18 — *выход за пределы допустимых значений* — может быть зарегистрирована как результат сравнения значений операндов команды RANGE-CHECK (см. гл. 15).

Ошибка типа 19 — *недостаточный объем памяти* — возникает, если память размером AM (см. гл. 15), имеющаяся в распоряжении процесс-машины, недостаточна для объекта, которому машина пытается ее динамически выделить, или если машина не способна получить необходимый объем физической памяти.

Ошибка типа 20 — *недопустимая обработка ошибок* — регистрируется в следующих случаях:

1) при попытке выполнить команду CONTINUE при наличии ошибки, не допускающей продолжение выполнения команд программы;

2) при попытке выполнения команды RAISE-FAULT с недопустимым кодом ошибки;

3) при попытке выполнения команд CONTINUE и TRANSFER-FAULT вне программ обработки ошибок.

Ошибка типа 21 — *имитация ошибки* — происходит, когда другая процесс-машина с помощью команды CONTROL-PROCESS-MACHINE имитирует для данной процесс-машины возникновение ошибки.

Ошибка типа 22 — *неопределенный доступ к операнду* — имеет место, когда содержимое ячейки, не являющейся операндом команды, используется для определения местоположения операнда, причем это содержимое имеет неопределенное значение. Это может произойти при обращении к ячейкам следующих типов:

1) ячейке «параметр», содержимое которой не было установлено в результате вызова текущего модуля по команде ACTIVATE;

2) ячейке «косвенный доступ к данным», используемой для переадресации к указателю, значение которого не определено;

3) ячейке «массив» с неопределенным значением индекса;

4) ячейке «указатель», содержащей косвенный потенциальный адрес, определяемый в свою очередь через указатель с неопределенным значением.

ПРОГРАММА ОБРАБОТКИ ОШИБОК

Каждому типу перечисленных выше ошибок в рассматриваемой вычислительной системе присвоен определенный номер. Этот номер соответствует некоторому двоичному разряду в *поле кода ошибки*, находящемся в заголовке модуля. Например, ошибке типа 1 (неверный код операции) соответствует разряд 1 в поле кода ошибки. Содержимое этого разряда, равное 1, указывает на необходимость обработки для данного модуля ошибок типа «неверный код операции». Нулевой (первый по порядку) двоичный разряд поля кода ошибки в заголовке модуля используется для указания на необходимость обработки ошибок типа 28—255, т. е. программно-определяемых ошибок (см. описание команды RAISE-FAULT).

При появлении ошибки некоторого типа система пытается вызвать программу обработки ошибок текущего модуля данного процесса. (По соглашению команды этой программы располагаются, начиная с первой команды модуля). Программа обработки ошибок будет вызвана, если разрешена обработка ошибок данного типа (содержимое соответствующего разряда поля кода ошибки равно 1). В противном случае система пытается вызвать другую программу обработки ошибок, для чего просматривает в обратном порядке стек активных модулей данно-

го процесса до отыскания такого модуля, в котором обработка ошибок данного типа разрешена. Если такой модуль найти не удастся, система уничтожает данную процесс-машину.

Предполагается (хотя это и не обязательное требование), что первым вызываемым модулем процесса является специальный модуль, генерируемый компилятором или операционной системой, в котором разрешена обработка ошибок всех типов.

Вызов программы обработки ошибок происходит как вызов внутренней процедуры. Поэтому такая программа имеет возможность работы с адресным пространством того модуля, которому принадлежит. (При этом запись активации соответствует последнему вызову этого модуля в данной процесс-машине.) При передаче управления модулю система передает ему также следующие пять параметров:

- 1) целое число, равное номеру типа ошибки;
- 2) указатель модуля, при выполнении которого обнаружена ошибка (в коде доступа указателя имеется разрешение на копирование);
- 3) указатель точки входа модуля, с которой началось его выполнение (в коде доступа этого указателя имеется разрешение на копирование);
- 4) ячейку типа «поле токенов», длина которой равна 5, а содержимое — адрес команды, вызвавшей состояние ошибки;
- 5) ячейку типа «поле токенов», длина которой равна 6.

Программе обработки ошибок разрешено только читать передаваемые ей фактические параметры. Значение последнего из них зависит от типа встретившейся ошибки. В частности, для ошибки «недействительный модуль» в этом параметре передается код типа ошибки (если причиной этой ошибки не является команда RAISE-FAULT). При наличии ошибки «имитация» в указанном параметре находится код, заданный в команде CONTROL-PROCESS-MACHINE, которая вызывает появление ошибки этого типа (если причиной ее появления не является команда RAISE-FAULT). При обнаружении ошибки «трассировка» в данном параметре передается информация о событии, вызвавшем это состояние. В остальных случаях в указанном параметре находятся первые 6 токенов команды, при выполнении которой обнаруживается ошибка.

В системе имеются также команды, позволяющие динамически разрешать и отменять в пределах модуля обработку ошибок отдельных типов; генерировать в программе в явной форме состояния ошибки; выходить из программы обработки ошибок с продолжением выполнения команд, следующих за командой, вызвавшей появление ошибки; повторять выполнение такой команды или передавать полномочия по обработке ошибки программе более высокого уровня.

Ошибки, возникающие во время работы программы обработки ошибок, обрабатываются так же, как и любые другие ошибки. Различие заключается лишь в том, что с целью предотвращения бесконечных повторных входов в указанную программу поиск подходящей программы обработки ошибок начинается с модуля, вызвавшего модуль, содержащий программу обработки ошибок, при выполнении которой регистрируется ошибка. Процесс обработки ошибок имеет структуру вложенных процедур: если ошибка обнаружена во время работы программы обработки ошибок А и обрабатывается программой обработки ошибок В, которая по команде RETURN или CONTINUE возвращает управление программе А, то последняя оказывается в своем предшествующем состоянии (т. е. параметры, передававшиеся программе А, оказываются вновь доступными ей в неизменном, первоначальном виде).

Будучи совокупностью команд, программа обработки ошибок должна начинаться с первой команды модуля. Поэтому в начале модуля располагаются либо команды указанной программы (начиная с команды LACT), либо команда перехода к ней.

СОСТОЯНИЕ ПРОЦЕССА ПОСЛЕ ОБНАРУЖЕНИЯ ОШИБКИ

Особого внимания при организации системы обработки ошибок заслуживает состояние процесса в момент обнаружения ошибки. В большинстве случаев команда, при выполнении которой обнаруживается ошибка, не влияет на состояние процесса. Программа обработки ошибок завершает свою работу одной из следующих четырех команд:

- 1) LOCAL RETURN — передача управления на повторное выполнение команды, обнаружившей ошибку;
- 2) CONTINUE — передача управления команде, которая выполнялась бы следующей, если бы не была обнаружена ошибка;
- 3) RETURN — уничтожение записи активации данного и всех последующих модулей с передачей управления модулю, который вызвал данный;
- 4) TRANSFER-FAULT — завершение работы данной программы обработки ошибок, инициирование поиска и передача управления аналогичной программе более высокого уровня.

Из этих общих правил имеется несколько исключений:

- 1) при выдаче команды LRETURN для выхода из состояния обработки ошибок, инициированного командой RAISE-FAULT, управление передается команде, следующей за командой RAISE-FAULT.

2) если ошибка регистрируется при обработке ячеек «поле», «запись» или «массив», то элементы этих ячеек, обработанные до обнаружения ошибки, получают новые значения, а значения остальных элементов не изменяются.

3) при выполнении в программе обработки ошибок команды CALL во избежание превращения стека активации в дерево, т. е. запоминающее устройство древовидной структуры, система ликвидирует все записи активации, расположенные в стеке активации под текущей записью.

НАБОР КОМАНД СИСТЕМЫ SWARD

В этом разделе рассматривается основное назначение команд системы SWARD. Детальное описание каждой команды дано в гл. 15.

КОМАНДЫ ОБЩЕГО НАЗНАЧЕНИЯ

В рассматриваемой системе имеются четыре команды общего назначения: MOVE, CONVERT, UNDEFINE и DISPLAY-OPERAND TYPE. Операндами этих команд могут быть ячейки, содержащие простые скалярные величины, массивы, строки, а также представляющие так называемые *срезы*, поля или записи. Команда MOVE используется для пересылки значения одного операнда в ячейку другого операнда. Команда CONVERT выполняет те же действия, что и команда MOVE, а также функцию преобразования данных. Например, при попытке использования команды MOVE для пересылки символьных данных в ячейку, предназначенную для целого числа, эта команда не выполняется и регистрируется ошибка «несовместимые операнды». Если же воспользоваться командой CONVERT, то указанная задача будет решена: символьные данные преобразуются в целое число в соответствии с заданными правилами преобразования.

Команда UNDEFINE используется для установки признака неопределенности значения операнда. Команда DISPLAY-OPERAND-TYPE позволяет получать информацию об атрибутах операнда. Этой командой можно, в частности, пользоваться для определения текущих атрибутов ячеек с динамически определяемыми типом, размером и границами.

АРИФМЕТИЧЕСКИЕ КОМАНДЫ

В эту группу входят команды: ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, ABSOLUTE, COMPLEMENT (по этой команде выполняется операция только над знаком операнда).

Каждая из первых пяти команд имеет два операнда; результат операции размещается на месте первого операнда. Команды **ABSOLUTE** и **COMPLEMENT** содержат по одному операнду. Операнды могут быть скалярами или массивами с числовыми значениями.

КОМАНДЫ СРАВНЕНИЯ И ПЕРЕХОДА

Формат команд **EQUAL-BRANCH-FALSE**, **NOT-EQUAL-BRANCH-FALSE**, **LESS-THAN-BRANCH-FALSE**, **GREATER-THAN-BRANCH-FALSE**, **LESS-THAN-OR-EQUAL-BRANCH-FALSE** и **GREATER-THAN-OR-EQUAL-BRANCH-FALSE** предусматривает наличие двух операндов и адрес команды. Выполнение этих команд сводится к сравнению значений обоих операндов. Если результат сравнения отрицательный, то управление передается по указанному адресу команды. В общем случае операндами этих команд может быть содержимое ячеек любого типа (например, указатели, строки символов, массивы, записи). К этой же группе принадлежат команды: **DEFINED-BRANCH-FALSE**, **ITERATE**, **ITERATE-REVERSE** и **CASE**. Первая из них служит для проверки, определено ли значение операнда. Две другие команды предназначены для организации циклов, а последняя команда — для передачи управления при наличии многих направлений, осуществляемой в зависимости от значения целого или так называемого порядкового числа.

ЛОГИЧЕСКИЕ КОМАНДЫ

К логическим командам относятся команды **AND**, **OR** и **NOT**. Операндами этих команд могут быть порядковые числа или массивы таких чисел.

КОМАНДЫ ПОИСКА И РАБОТЫ СО СТРОКАМИ ДАННЫХ

В то время как многие машинные команды могут работать с операндами, представляющими собой строки символов, существуют специальные команды, выполняющие операции только над данными такого типа. Операндами этих команд могут быть поля или строки символов либо токенов.

По команде **CONCATENATE** значение одного операнда присоединяется к концу значения другого операнда. Команда **MOVE-SUBSTRING** позволяет часть строки, выделенную в одном операнде, поместить на место части строки другого операнда. С помощью команды **INDEX** в указанной строке может быть

проведен поиск заданной части строки. По команде LENGTH можно определить текущую длину строки.

К этой же группе команд относится команда SEARCH. Эта команда позволяет в массиве или в срезе массива найти элемент с заданным значением (определить индексы этого элемента).

КОМАНДЫ УПРАВЛЕНИЯ

Команды управления используются для безусловной передачи управления: для организации передачи управления между модулями используются командами CALL, ACTIVATE и RETURN, для организации обращения к внутренним подпрограммам данного модуля — командами LOCAL-CALL, ACTIVATE и LOCAL-RETURN, для изменения порядка выполнения команд в модуле — командой BRANCH.

В команде CALL задается имя элемента, которому передается управление (точка входа в модуль), и список фактических параметров. Для части этих параметров может быть определен признак того, что они могут только читаться, т. е. вызываемый модуль не может их изменить, их освободить их. Команда CALL распределяет для переменных память, отводимую в динамической части адресного пространства вызываемого модуля, и передает управление в заданную точку входа. Если вызываемый модуль должен извне получать параметры, то до обращения к ним в этом модуле должна быть выдана команда ACTIVATE. В команде определяется список получаемых параметров. При выполнении этой команды проверяется соответствие между передаваемыми фактическими параметрами и принимающими формальными, а также осуществляется инициализация последних (передаются адреса параметров). По команде RETURN освобождается память динамической части адресного пространства, и управление возвращается вызывающему модулю.

В команде LCALL¹⁾ в форме адреса команды задаются адрес внутренней процедуры и список фактических параметров. Если процедура получает значения параметров извне, то обращению к ним должна предшествовать команда ACTIVATE. По команде LCALL не выполняется какого-либо динамического распределения памяти, т. е. потребности внутренних процедур обеспечиваются средствами программного обеспечения не в полной мере. Поэтому функции выделения памяти и определения области действия имен переменных при необходимости

¹⁾ Сокращенное название программы LOCAL-CALL. Аналогично полным названием команды LRETURN является LOCAL-RETURN. Список сокращений приведен в описаниях команд в гл. 15. — *Прим. перев.*

возлагаются на компилятор. По команде LRETURN производится возврат управления из данной процедуры команде, следующей за последней выполнявшейся командой LCALL.

КОМАНДЫ АДРЕСАЦИИ

Эта группа команд связана с операциями над указателями, потенциальными адресами и объектами системы. Команда COMPUTE-CAPABILITY для указанного операнда формирует его потенциальный адрес, команда COMPUTE-INDIRECT-CAPABILITY позволяет получить косвенный потенциальный адрес для заданного указателя. Команда CHANGE-ACCESS предназначена для сужения прав доступа (включения дополнительных ограничений в код доступа) в потенциальном адресе. С помощью команды CHANGE-LOGICAL-ADDRESS переименовывают существующий объект, т. е. назначают новый логический адрес для объекта. Команда COMPUTE-CAPABILITY-EXTERNALLY позволяет сформировать потенциальный адрес для указанного операнда из другого модуля, возможно, даже с соответствием с записью активации, связанной с другой процесс-машиной. Эта команда предназначена прежде всего для отладки программ.

По команде ALLOCATE создается объект «память данных», предназначенный для ячейки определенного типа на случай ее возможного создания в процессе работы системы. По команде DESTROY можно уничтожить объект любого типа.

Команда CREATE-MODULE позволяет на основании внешнего модуля получить объект «модуль». С помощью команды COMPUTE-ENTRY-CAPABILITY можно рассчитать потенциальный адрес точки входа указанного модуля. Команда LINK присваивает значение указателю в некотором объекте «модуль». Команды COMPUTE-ENTRY-CAPABILITY и LINK используются для организации взаимной связи модулей программы.

Если воспользоваться командой DESCRIBE-CAPABILITY, а в качестве операнда задать указатель, можно получить ряд параметров соответствующего потенциального адреса и самого адресуемого объекта.

КОМАНДЫ УПРАВЛЕНИЯ ПРОЦЕСС-МАШИНАМИ

Эта группа команд тесно связана с понятием процесс-машин системы.

Команда CREATE-PROCESS-MACHINE позволяет создать процесс-машину, передать ей требуемый объем имеющейся в наличии памяти и начать ее выполнение с заданной точки входа модуля. TRANSFER-AM позволяет перераспределить

объем памяти между двумя процесс-машинами. С помощью команды **CONTROL-PROCESS-MACHINE** одна процесс-машина может останавливать и запускать другую процесс-машину, а также формировать условия появления ошибки в другой процесс-машине или изменять приоритет последней. Посредством команды **COMPUTE-PROCESS-MACHINE-CAPABILITY** можно получить потенциальный адрес процесс-машины, обрабатывающей данную команду. При использовании команды **DELAY** приостанавливается выполнение данной процесс-машины на указанный интервал времени.

Команды **GUARD** и **UNGUARD** позволяют защищать критические области в последовательностях команд от одновременной обработки несколькими процесс-машинами, что облегчает построение мониторов.

Команды **CREATE-PORT**, **SEND** и **RECEIVE** используются для организации связи между процессами по данным. С помощью команды **SEND** сообщение может быть послано в порт, а с помощью команды **RECEIVE** — получено из него.

команды отладки

Эта последняя группа команд связана с отладкой программ и обработкой ошибок. С помощью команд **ENABLE** и **DISABLE** программа может динамически устанавливать и отменять регистрацию ошибок определенного типа, которые подлежат программной обработке ошибок данного модуля. Посредством команды **RAISE-FAULT** можно явно запросить регистрацию ошибки и инициировать работу обработчиков ошибок. По команде **RANGE-CHECK** можно задать диапазон значений операнда и условие генерирования ошибки, если текущее значение операнда выходит за пределы указанного диапазона.

Включение команды **CONTINUE** в обработчик ошибок позволяет вернуться к выполнению команд модуля, в котором была зарегистрирована ошибка, а именно передать управление команде, непосредственно следующей за той, которая привела к ошибке. (Если требуется повторить команду, вызывающую возникновение ошибки, следует воспользоваться командой **LRETURN**.) Если во время работы данной программы обработки ошибок появляется необходимость выполнять обработку с помощью программы более высокого уровня, то соответствующая передача управления может быть выполнена с помощью команды **TRANSFER-FAULT**.

Команда **TRACE** позволяет установить в указанном модуле режим трассировки команд перехода, команд передачи управления модулям и внутренним процедурам и (или) команд **MARKER**. В таком случае выполнение команд отмеченных типов будет сопровождаться регистрацией состояния «трассиров-

ка». Соответственно выполнение команды **MARKER** либо не будет ни в чем проявляться, либо будет сопровождаться индикацией состояния «трассировка», если режим маркерной трассировки разрешен соответствующими параметрами команды **TRACE**.

ПРИМЕРЫ СТРУКТУРЫ МОДУЛЯ

В целях иллюстрации особенностей архитектуры системы **SWARD** в данном разделе рассматриваются режимы компиляции и выполнения нескольких программ. Текст первой программы на языке ПЛ/1 соответствует второй программе для учебной ПЛ-машины (гл. 6), использовавшейся также для иллюстрации архитектуры **SYMBOL** (гл. 9). Исходный текст программы на языке ПЛ/1 имеет следующий вид:

```

1 (SUBSCRIPTRANGE): XYZ: PROCEDURE;
2 DECLARE A(4) FLOAT DECIMAL(7);
3 DECLARE (B, C) FIXED BINARY;
4 B=4;
5 A=1;
6 C=B+B*B;
7 CALL ZZZ(B);
8 A(B)=C;
9 ZZZ: PROCEDURE(M);
10 DECLARE M FIXED BINARY;
11 C=M+M;
12 END;
13 END;
```

Требуется сформировать строку токенов, которая содержит модуль¹⁾, соответствующий этой программе, и проанализировать его выполнение. Естественно было бы начать со структуры заголовка модуля, однако большинству его полей нельзя присвоить значения до окончания формирования всего модуля. Поэтому построение заголовка будет временно отложено в предположении, что размер модуля невелик (в отношении как данных, так и команд). Это позволяет принять значения **CAS** и **IAS** равными 2 (для записи адресов ячеек и команд отводится 2 токена памяти).

Прежде всего обратимся к адресному пространству модуля. Все переменные рассматриваемой программы подлежат включению в динамическую область адресного пространства. Адресное пространство модуля имеет следующий вид:

01	70111000B000004D7XXXXXX	Array A(floating point)
18	F800000	Integer B
1F	F800000	Integer C
26	6005FXXXXXXX	Parameter integer M

¹⁾ По установленной терминологии **SWARD** — внешний модуль. — *Прим. перев.*

В левой колонке указаны адреса всех ячеек. Поскольку в исходном тексте программы отсутствует присвоение переменным начальных значений, в соответствующих ячейках адресного пространства указан признак неопределенного значения.

Далее перейдем к рассмотрению машинных команд для анализируемой программы. Для простоты воспользуемся представлением машинных команд на некотором условном языке типа языка ассемблера. Рядом с каждой командой, записанной на этом языке, будут приводиться соответствующая машинная команда и ее адрес (в форме адреса команды).

Первой командой в модуле является команда **ACTIVATE**. В ней количество адресных полей непостоянно. Этим полям предшествует поле так называемых непосредственных данных, в котором задается число параметров. В остальные поля записываются адреса параметров в форме адресов ячеек. Команда **ACTIVATE** присваивает параметрам начальные значения и проверяет, идентичны ли типы каждого подобного формального параметра и соответствующего ему фактического параметра. Для рассматриваемой программы эта команда имеет вид

ACT 0 (01: C00)

Здесь **C00** — запись данной команды в шестнадцатеричном коде, **C** — код операции, а **01** — адрес команды. Величина **00** в поле непосредственного адреса означает отсутствие передаваемых в данный модуль параметров.

После компилирования программы и ее загрузки полученный модуль (с именем **XYZ**) будет вызываться некоторым другим модулем (например, операционной системой). Именно в этот момент формируется запись активации, выполняется копирование в нее динамической части адресного пространства и распределение памяти для элементов массива **A**. Однако команды модуля **XYZ** могут обращаться к ячейкам так, как если бы они были физически размещены внутри данного модуля. Если адресуемая ячейка принадлежит динамической части адресного пространства, машина автоматически преобразует этот адрес в адрес соответствующего элемента записи активации.

После операторов описания в исходной программе расположен оператор **B=4**, реализуемый командой **MOVE**. В этой команде используется литерал, поскольку присваиваемое **B** значение **4** является целым числом, которое меньше **10**. Команда формируется в виде¹⁾

MOVE B, '4' (04: 118004)

¹⁾ Некоторому различию записи литерала в команде на условном языке ассемблера в данном тексте и в столбце комментариев к приводимому ниже полному тексту внешнего модуля не следует придавать значения. — *Прим. перев.*

где 18 — адрес ячейки В, а 004 — литерал. После выполнения данной команды содержимое ячейки В будет иметь значение, равное F000004.

Оператор исходной программы A=1 (пятая строка текста) будет представлен следующей машинной командой:

MOVE A, '1' (0B: 10100F001)

Величина 0100F в коде машинной команды является адресом массива А, причем 01 — адрес ячейки, а 00F — код знака *, играющего роль индекса. Следует обратить внимание на то, что в данном случае скалярная величина пересылается в массив. Кроме того, эта величина является целым числом, а элементы массива — десятичными числами с плавающей точкой. Системой учитываются оба этих обстоятельства, и в каждый элемент массива А помещается десятичное число с плавающей точкой, равное 1.

Для шестой строки исходного текста — оператора $C = B + B * B$ — генерируются коды:

MOVE C, B (13:11F18)

MULT C, B (18:41F18)

ADD C, B (1D:21F18)

Каждая из этих команд проверяет, определено ли значение исходного операнда, являются ли оба операнда арифметическими (т. е. числами), достаточно ли по своим размерам место, куда должен быть записан результат операции.

Седьмая строка исходного текста — оператор CALL. Для него генерируется следующая команда¹⁾:

LCALL ZZZ, 1, RW, B (22: 000B??01318)

В этой команде 000B — код операции. Следующее поле в данной команде должно содержать адрес вызываемой в этом модуле внутренней процедуры. Значение этого адреса пока еще не определено. Содержимое следующего поля, имеющего длину 2 токен и называемого полем непосредственных данных, указывает количество фактических параметров. Остальная часть команды содержит информацию об адресах фактических параметров и формах доступа к каждому из них (только чтение или чтение — запись). В данном случае передается один параметр, к которому применимы операции чтения и записи. Команда LCALL, служащая для обращения к внутренней процедуре, в этом месте программы приостанавливает выполнение после-

¹⁾ Знак «?» обозначает временно не определенные части кода, получающие значения после формирования последующих частей модуля, а символы RW указывают санкционируемые возможности чтения и записи. — *Прим. перев.*

довательности команд и передает управление команде с указанным адресом.

Кодирование следующего оператора, расположенного в восьмой строке исходной программы, иллюстрирует возможности, предоставляемые архитектурой системы SWORD для работы с элементами массивов. В данном случае формируется следующая команда:

MOVE ;A(B),C (2D: 101181F)

Здесь 0118 — адрес элемента массива, 01 — адрес ячейки массива A, а 18 — адрес ячейки, содержимое которой используется в качестве единственного индекса элементов этого одномерного массива. Отметим, что если при выполнении данной команды текущее значение B оказалось бы вне диапазона допустимых значений индекса A, то была бы зарегистрирована ошибка «выход за пределы допустимых значений».

Поскольку оператор в девятой строке исходной программы является началом внутренней процедуры, необходимо «обойти» оператор этой процедуры. Для этого компилятор генерирует команду

B ? (34: E??)

(До окончания процесса компилирования адрес перехода остается временно не определенным.) Следующая генерируемая команда имеет вид

ACT 1,M (37: C0126)

Эта команда задает количество параметров равным 1. При ее выполнении проверяется соответствие типов формального и передаваемого фактического параметров и присваивается начальное значение формальному параметру для возможности последующих обращений к нему как к фактическому параметру.

Для оператора, расположенного в 11-й строке исходного текста, формируются команды

MOVE C,M (3C: 11F26)

ADD C,M (41: 21F26)

Машина распознает, что операнды-источники в этих командах по своему типу являются параметрами, автоматически выполняет переадресацию к фактическому параметру и использует его значения в качестве значений этих операндов-источников.

Программный модуль завершается командами

LRETURN (46:000C)

RETURN (4A:0A)

По команде LRETURN (ВОЗВРАТ ИЗ ВНУТРЕННЕЙ ПРОЦЕДУРЫ) управление возвращается команде, следующей за последней выполнявшейся командой LCALL (в данном случае управление получает команда с адресом 2D). Можно заметить, что адресом перехода в рассматривавшейся выше команде В должен быть адрес команды RETURN. По команде

Смещение		Комментарий
001	0001E0004F0009A	Header Indices
015	220000000	CAS/IAS/SIS/Faults
01E -01	70111000B000004D7XXXXXX	A (floating-point array)
18	F800000	B (integer)
1F	F800000	C (integer)
26	6005FXXXXXX	M (parameter integer)
04F 01	C00	ACT 0
	118004	MOVE B,4
	10100F001	MOVE A,1
	11F18	MOVE C,B
	41F18	MULT C,B
	21F18	ADD C,B
	000B3701318	LCALL ZZZ,1,RW,B
	101181F	MOVE A(B),C
	E4A	B %A
37	C0126	ZZZ: ACT 1,M
	11F26	MOVE C,M
	21F26	ADD C,M
	000C	LRET
4A 0A		RETURN

Рис. 14.4. Внешний модуль XYZ.

RETURN выполнение этого модуля завершается и управление передается вызывавшему его модулю.

Полный текст модуля приведен на рис. 14.4. Модуль представлен не в виде единого кода по всей совокупности смежных токенов, а с выделением в строках отдельных ячеек и команд

Таблица 14.2. Параметры программ для машин разного типа

Машина	Количество выполненных команд	Размер объектного модуля, байт	Общий размер программы, байт
Учебная ПЛ-машина	62	96	369
SYMBOL	81	324	396
Система 370	117 ¹⁾	392	5536
SWARD	14	37	77

¹⁾ И еще ~90 000 команд в начале выполнения программы для выделения памяти подсистеме организации связи с подпрограммами.

для более наглядного рассмотрения. Первые две колонки не входят в структуру модуля. В них записаны порядковый номер первого токена соответствующей строки (от начала модуля) и порядковый номер первого токена строки в адресном пространстве или области команд. В каждой строке имеется также текст комментария, не являющийся, конечно, частью модуля.

```
TESTEST: PROCEDURE;
DECLARE SIZE FIXED DECIMAL(4);
DECLARE 1 B(7),
        2 N CHAR(8),
        2 T CHAR(2),
        2 A POINTER;
DECLARE UNNAME CHAR(8) INIT('XXXXXXXX'),
        CODE FIXED DECIMAL(1) INIT(9);
DECLARE NULL BUILTIN;
B(1).N='ABCDEFGH';
B(1).T='ER';
B(1).A=NULL;
B(2).N='ABCDEFGH';
B(2).T='MD';
B(2).A=ADDR(UNNAME);
SIZE=2;
CALL MATCHES(B,UNNAME,CODE,SIZE);
END;
```

Рис. 14.5. Процедура
TESTEST на языке
ПЛ/1.

```
with PILLOW; use PILLOW;
procedure TESTEST is
  SIZE : CURRENT_ENTRIES;
  B     : MATCH_TABLE (INTEGER 1..7);
  UNNAME : PROGNAME := "XXXXXXXX";
  CODE   : MATCH_ERROR := 9;
begin
  B(1).NAME := "ABCDEFGH";
  B(1).ETYPE := "ER";
  B(1).ADDRESS := null;
  B(2).NAME := "ABCDEFGH";
  B(2).ETYPE := "MD";
  B(2).ADDRESS := UNNAME'ADDRESS; — Not standard Ada
  SIZE := 2;
  MATCHES(B,UNNAME,CODE,SIZE);
end TESTEST;
```

Рис. 14.6.
Процедура
TESTEST на
языке Ада.

В табл. 14.2 продолжено сопоставление параметров программ: программы для учебной ПЛ-машины, эквивалентной ей программы на языке SPL для системы SYMBOL и рассматривавшейся в данной главе программы на языке ПЛ/1 для Системы 370 и машины SWARD.

В качестве еще одного примера рассмотрим программу, состоящую из двух отдельно компилируемых процедур (рис. 14.5—14.8). Программа написана на двух языках — Ада и


```

MATCHES: PROCEDURE(BODY,UNRESNAME,MATCHCODE,SIZE);
DECLARE 1 BODY(*),
        2 NAME CHAR(8),
        2 TYPE CHAR(2),
        2 ADDRESS POINTER;

DECLARE
    MODULE CHAR(2) STATIC INIT('MD'),
    ENTRYPT CHAR(2) STATIC INIT('EP'),
    EXTREF CHAR(2) STATIC INIT('ER');

DECLARE NULL BUILTIN;
DECLARE MATCHCODE FIXED BINARY;
DECLARE UNRESNAME CHAR(8);
DECLARE SIZE FIXED BINARY;
DECLARE
    I FIXED BINARY;
    J FIXED BINARY;
MATCHCODE=2;
IF (SIZE>0)&(SIZE->2000)
THEN
DO;
    MATCHCODE=0;
    DO I=1 TO SIZE WHILE(MATCHCODE=0);
        IF BODY(I).ADDRESS=NULL
        THEN DO;
            MATCHCODE=1;
            DO J=1 TO SIZE WHILE(MATCHCODE=1);
                IF (BODY(I).NAME=BODY(J).NAME)&
                    ((BODY(J).TYPE=MODULE)|
                     (BODY(J).TYPE=ENTRYPT))
                THEN DO;
                    MATCHCODE=0;
                    BODY(I).ADDRESS=BODY(J).ADDRESS;
                END;
            ELSE;
            END;
        IF MATCHCODE=1 THEN UNRESNAME=BODY(I).NAME;
        ELSE;
        END;
    ELSE;
    END;
END;
ELSE;
END;

```

Рис. 14.7. Процедура MATCHES на языке ПЛ/1.

ПЛ/1. Приведенный пример позволяет проиллюстрировать многие возможности архитектуры: работу со строками символов, записями, массивами, указателями или переменными доступа, записями, компонентом которых является массив записей, а также оформление вызовов подпрограмм и работу с динамической и статической частями адресного пространства.

Внешний модуль для процедуры TESTEST представлен на

```

package PILLOW is
  type PROGNAMЕ      is STRING(1..8);
  type MATCH_ERROR   is INTEGER range 0..2;
  type CURRENT_ENTRIES is INTEGER range 0..2000;
  type MATCH_TABLE_ENTRY is
    record
      NAME      : PROGNAMЕ;
      ETYPE     : STRING (1..2);
      ADDRESS   : access PROGNAMЕ;
    end record ;
  type MATCH_TABLE is array (INTEGER range <>) of MATCH_TABLE_ENTRY;
  MODULE : constant STRING := "MD";
  ENTRYPT : constant STRING := "EP";
  EXTREF : constant STRING := "ER";
  procedure MATCHES
    (BODY      : in out MATCH_TABLE;
     UNRESNAME : out PROGNAMЕ;
     MATCHCODE : out MATCH_ERROR;
     SIZE      : in CURRENT_ENTRIES);
end PILLOW;

package body PILLOW is
  procedure MATCHES
    (BODY      : in out MATCH_TABLE;
     UNRESNAME : out PROGNAMЕ;
     MATCHCODE : out MATCH_ERROR;
     SIZE      : in CURRENT_ENTRIES) is
  begin
    MATCHCODE := 2;
    if (SIZE > 0) and (SIZE <= 2000) then
      MATCHCODE := 0;
      for I in 1..SIZE loop
        exit when MATCHCODE /= 0;
        if BODY(I).ADDRESS = null then
          MATCHCODE := 1;
          for J in 1..SIZE loop
            exit when MATCHCODE /= 1;
            if (BODY(I).NAME = BODY(J).NAME) and
              ((BODY(J).ETYPE = MODULE) or
               (BODY(J).ETYPE = ENTRYPT)) then
              MATCHCODE := 0;
              BODY(I).ADDRESS = BODY(J).ADDRESS;
            end if ;
          end loop ;
          if MATCHCODE = 1 then
            UNRESNAME = BODY(I).NAME;
          end if ;
        end if ;
      end loop ;
    end if ;
  end MATCHES;
end PILLOW;

```

Рис. 14.8. Процедура MATCHES на языке Ада.

Смещение		Комментарий
01	0001E00071000AB0012B	Header
15	330000000	CAS/IAS/SIS/FC
1E 001	E40F0000	SIZE
009	702C100290000007801D0029	B (array of records)
	0000B008	B.N (cf component) cdo=8
	0010B002	B.T (cf component) cdo=10
	00149	B.A (pointer comp) cdo=18
	000000	
03B	B008E7E7E7E7E7E7E7E7	UNNAME (character field)
04F	E1009	CODE
71 054	900000000000000000	MATCHES (pointer)
	000000	
06A	B008C1C2C3C4	'ABCDEFGH'
	C5C6C7C8	
07E	B002C5D9	'ER'
086	B002D4C4	'MD'
AB 001	C00	ACT 0
	1009000100806A	MOVE B(1).N, 'ABCDEFGH'
	1009000101007E	MOVE B(1).T, 'ER'
	00010090001018	UNDEF B(1).A
	1009000200806A	MOVE B(2).N, 'ABCDEFGH'
	10090002010086	MOVE B(2).T, 'MD'
	0010009000201803B	CCAP B(2).A, UNNAME
	10010002	MOVE SIZE, 2
	D054043009000F	CALL MATCHES, 4, R/W(B), R/W(UNNAME),
	303B304F3001	R/W(CODE), R/W(SIZE)
	0A	RETURN

Рис. 14.9. Внешний модуль TESTEST.

рис. 14.9. Поскольку оба модуля по размерам не слишком велики, оказалось возможным ограничиться адресами ячеек и команд длиной 3 токен. В связи с тем что В является массивом записей или структур, он представлен ячейкой «массив», а его вложенный тег соответствует записи, содержащей три компонента.

Модуль MATCHES, указатель и несколько констант размещены в статической части адресного пространства. Предполагается, что некоторая программа (например, редактор связей) посредством команды LINK будет инициализировать выполнение модуля MATCHES, пользуясь потенциальным адресом точки его входа.

Структура получающегося кода достаточно очевидна. Определенный интерес может представлять рассмотрение одной из команд MOVE как примера с адресами в виде записей.

Исходному тексту MATCHES, представленному на рис. 14.7 и 14.8, соответствует внешний модуль, приведенный на рис. 14.10. Коды внешнего модуля снабжены адресами и комментариями.

Смещение		Комментарий
001	0001E0008A000A9001B4	Header
015	330000000	CAS/IAS/SIS/Faults
01E 001	6030702C100000000000	BODY (parameter array of records) (D-bounded)
	801D0029	
	0000B008	BODY.NAME (cf component) cdo=8
	0010B002	BODY.TYPE (cf component) cdo=10
	00149	BODY.ADDRESS (cf component) cdo=18
	0000000	
038	6005F00000000	MATCHCODE (integer parameter)
044	6008B00800000000	UNRESNAME (cf parameter)
053	6005F00000000	SIZE (integer parameter)
05F	F800000	I
066	F800000	J
08A 06D	B002D4C4	MODULE (character field)
075	B002C5D7	ENTRYPT (character field)
07D	B002C5D9	EXTREF (character field)
085	F0007D0	'2000.'
0A9 001	C04001046038055	ACT 4,BODY,UNRESNAME, MATCHCODE,SIZE
	10380002	MOVE MATCHCODE,2
	9053000010A	GTBF SIZE,0,%H
	A05308510A	LEBF SIZE,2000,%H
	10380000	MOVE MATCHCODE,0
	105F0001	MOVE I,1
	A05F05310A	LEBF I,SIZE,%H
048	7038000010A	%A: EQBF MATCHCODE,0,%H
	000400105F018067	DEFBF BODY(I).ADDRESS,%B
	E100	B %G
067	10380001	%B: MOVE MATCHCODE,1
	10660001	MOVE J,1
	A0660530E8	LEBF J,SIZE,%F
081	703800010E8	%C: EQBF MATCHCODE,1,%F
	700105F008	EQBF BODY(I).NAME,
	0010660080DE	BODY(J).NAME,%E
	600106600806D0C3	NEBF BODY(J).NAME,MODULE,%D
	70010660080750DE	EQBF BODY(J).NAME,ENTRYPT,%E
0C3	10380000	%D: MOVE MATCHCODE,0
	100105F018	MOVE BODY(I).ADDRESS,
	001066018	BODY(J).ADDRESS
ODE	5066053081	%E: ITERATE J,SIZE,%C
OE8	70380001100	%F: EQBF MATCHCODE,1,%G
	104400105F008	MOVE UNRESNAME,BODY(I).NAME
100	505F053048	%G: ITERATE I,SIZE,%A
10A 0A		%H: RETURN

Рис. 14.10. Внешний модуль MATCHES.

ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ СИСТЕМЫ SWARD

Основной отличительной особенностью, определенной уникальностью архитектуры системы SWARD, является целенаправленность всех ее средств на разрешение традиционных проблем программного обеспечения вычислительных систем. Главные пути достижения этой цели анализируются ниже.

Рассматриваемая система осуществляет в значительной мере семантический контроль обрабатываемой информации, что должно способствовать значительному увеличению эффективности процессов тестирования и отладки программ и ограниченному последствию ошибок, допущенных на этапе создания рабочих программ.

Вследствие того что архитектура SWARD основана на понятиях объектов и потенциальной адресации, оказывается возможным достичь высокой степени единообразия в решении вопросов программирования. Ко всем объектам, входящим в архитектуру (модулям, процесс-машинам, портам, памяти данных), а также к устройствам ввода-вывода без памяти можно адресоваться аналогичным образом. Это позволяет существенно упростить как системное программное обеспечение, так и программы пользователей. Например, традиционным системам присуще многообразие средств установления связи между разными элементами. Так, для установления связи между программными модулями используется редактор связи, для связи программ с файлами — операторы языка управления заданиями и макрокоманды открытия этих файлов в программах, что же касается системы SWARD, то здесь достаточно ограничиться одним общим понятием «привязка».

Положенный в основу системы SWARD принцип одноуровневой организации памяти может быть распространен и на структуру языков программирования, существенно сокращая потребность в использовании традиционных принципов организации ввода-вывода. Все это упрощает работу программиста, позволяя ему использовать единый подход к проблемам манипулирования данными.

Возможность пользоваться одними и теми же командами SEND и RECEIVE как основными командами ввода-вывода для работы с внешними устройствами без памяти и как командами обмена данными между процессами дает много преимуществ. Во-первых, это тоже способствует более единообразному стилю работы программиста с системой, поскольку становится несущественным способ пересылки данных (например, строки сим-

волов) — через порт на печатающее устройство, терминал или другому процессу. Таким образом, в системе достигается использование единого принципа передачи данных. Во-вторых, не внося изменений в программы, оказывается возможным заменять в операциях обмена внешние устройства на процессы и наоборот. В-третьих, работа системы передачи данных отличается синхронностью выполнения операций записи и чтения как в случае обмена типа процесс — процесс, так и в случае обмена процесс — внешнее устройство. В соответствии с этим для системы действительно только одно понятие параллельности выполнения операций — на уровне процессов. Понятие прерывания в данном случае отсутствует¹⁾.

К другим единым принципам организации вычислительной системы SWARD, позволяющим упростить процесс программирования и достичь его большего соответствия возможностям аппаратных средств системы, следует отнести введение специальных средств обработки ошибок, потенциальной формы адресации для совместного доступа к данным и для их защиты от несанкционированного доступа, системы команд, инвариантных к типу обрабатываемых данных, и отказ от привилегированного режима выполнения команд.

Эффективная система вызова подпрограмм, разбиение адресного пространства на небольшие области санкционированного доступа, средства обработки ошибок, одноуровневая память, использование команд GUARD, UNGUARD, SEND, RECEIVE и другие возможности системы благоприятствуют разработке хорошо структурированных программ на основе принципов модульности, обеспечению защиты информации от несанкционированного доступа и распараллеливанию процессов.

Все перечисленное выше касается программирования в целом, однако можно сделать ряд замечаний относительно орга-

¹⁾ Упоминаемые автором синхронность операций обмена и отсутствие прерывания (сравните со способом доступа с очередями в ОСЕС) удобны для программиста лишь до тех пор, пока его устраивают временные характеристики передачи данных. В противном случае предпочтительнее перейти на несколько более трудоемкий метод разработки программ (например, базисный способ доступа в ОСЕС), чтобы, используя те же технические средства — при обмене с внешними устройствами, физически построенном на асинхронизме и прерываниях, — обеспечить требуемые параметры функционирования создаваемого программного обеспечения.

Оценивая различные проявления униформизма в отношении объектов системы или связей между ними, следует иметь в виду, что хотя такой подход обеспечивает простоту разработки и реализации программных средств, они не определяют противоречия с принципом эффективного использования ресурсов системы. — *Прим. перев.*

низации работы компиляторов, операционных систем и систем доступа к базам данных. Благодаря использованию теговой памяти, прямому распознаванию типов данных с многоуровневой организацией, таких, как массивы и структуры, мощной системе команд, инвариантных к типу обрабатываемых данных, должны значительно снижаться затраты на разработку компиляторов и сложность последних.

Данная архитектура позволяет исключить многие традиционно сложные компоненты операционных и других управляющих систем, освобождая их от задач управления памятью, защиты данных, мониторинга функций по синхронизации процессов, организации их связи по данным и обработки прерываний.

Использование команд, инвариантных к типу обрабатываемых данных, и теговой памяти предполагает осуществление «привязки» команд и данных на самых поздних этапах подготовки программы к выполнению. Конкретное функциональное назначение команды определяется к моменту ее выполнения посредством информации в тегах и ячейках операндов команды. Это существенно для реализации работы различных систем управления базой данных в режиме независимости от типов данных.

К недостаткам данной архитектуры можно отнести как определенные ограничения, так и трудности ее реализации. Например, некоторые числовые ограничения на включенные в теги параметры данных следовало бы сделать менее жесткими (увеличить допустимый размер строк символов). В системе неявно зафиксировано наименьшее допустимое значение индексов массивов, равное 1. Было бы желательно предоставить пользователю самому устанавливать эту величину. К недостаткам можно также отнести отсутствие в системе прямых средств для установки начальных значений элементов массивов (см. упреждения в конце главы).

Оказалось, что реализация отдельных принципов архитектуры либо слишком дорогостояща, либо практически неэффективна. То же самое можно заметить и в отношении принципа непостоянного размера адресов ячеек и команд. Возникают трудности с реализацией операций десятичной арифметики с плавающей точкой. В отдельных случаях для достижения простоты реализации имеет смысл пожертвовать однородностью команд и их инвариантностью к типам обрабатываемых данных. В частности, является желательной возможность выполнять с помощью команды ADD (без необходимости дополнительного программного преобразования данных) сложение десятичных чисел с фиксированной и плавающей точками или прибавление

скалярной величины ко всем элементам массива. Однако от выполнения с помощью подобных команд деления десятичных чисел с плавающей точкой одного массива на десятичные числа с фиксированной точкой другого массива можно отказаться.

ОКРУЖАЮЩАЯ СРЕДА СИСТЕМЫ

Единственная существующая реализация архитектуры системы SWARD имеет так называемую горизонтальную структуру микропрограммного уровня машины¹⁾. Система функционирует на одном микропроцессоре, работающем в режиме разделения времени между всеми активными процесс-машинами. Планируется подключение к системе еще нескольких процессоров.

Чтобы дать общее представление о ресурсах, необходимых для построения системы, укажем, что микропрограмма для системы SWARD занимает ~5000 52-битовых слов памяти. В данной микропрограмме заложена возможность работы только с одним фиксированным значением длины адресов ячеек и команд (3 токен), не предусмотрена возможность работы с расширенным множеством команд и не обеспечиваются операции десятичной арифметики с плавающей точкой. По сравнению с другими командами большая часть микропрограмм приходится на команды MOVE (поскольку ее операндами могут быть ячейки любого типа, причем сложной структуры), CREATE-MODULE и DESTROY. Сложнее всего оказалась часть программы, реализующая команду DESTROY. Приходится учитывать, например, что уничтожение объекта «процесс-машина» может неявно повлечь за собой необходимость уничтожения ряда других объектов, в том числе и других процесс-машин.

Исходным языком программирования, поддерживаемым программным обеспечением в SWARD, является язык Soada — расширенная версия языка Ада. Необходимость модификации вызвана тем, что его структура не в полной мере соответствует принципам системы SWARD. Язык Ада разрабатывался специально для систем с традиционной архитектурой, для которой характерно установление всех связей между программными элементами выполняемого задания на ранних стадиях его обработки системой — на этапе компилирования. Что касается системы SWARD, то она почти по всем показателям может быть

¹⁾ Представление об определении горизонтальной структуры микропрограммного уровня и ее отличии от вертикальной структуры читатель может получить из книги Таненбаум Э., Многоуровневая организация ЭВМ. Пер. с англ. М.: Мир, 1979, с. 232—236. — *Прим. ред.*

охарактеризована как вычислительная среда с «отсрочкой» установления связей до самых последних стадий. По сравнению с языком Ада в языке Soada предусмотрено

1) использование переменных с инвариантными атрибутами доступа (динамически определяемым типом) и формальных параметров;

2) возможность включения кодов доступа в переменные доступа;

3) возможность рассмотрения пакетов и подпрограмм как объектов;

4) использование строк переменной длины;

5) обработка признаков типа на этапе выполнения.

Последняя возможность иллюстрируется следующими фрагментами операторов на языке Soada:

```
if X.all'OSTYPE=PACKAGE...
```

```
if A'VTYPE=INTEGER or A'VTYPE=COLOR...
```

Операционная система машины SWARD отличается тем, что лишь в незначительной степени занята выполнением функций традиционных операционных систем (управления процессами, памятью, обеспечением режима разделения времени между процессами и т. п.), так как эти функции заложены в основу машины SWARD. Согласно принципам архитектуры системы SWARD, операционную систему можно рассматривать как пакет прикладных программ, обеспечивающих при необходимости выполнение определенных функций обслуживания (прикладные программы не поддерживают непосредственной связи с операционной системой). Большая часть таких функций связана с управлением справочниками, в частности согласование системы потенциальной адресации с системой вводимых пользователем имен и кодов доступа.

К основному процессору системы SWARD подключено несколько микропроцессоров, функционирующих в основном в роли каналов ввода-вывода. Предусмотрено, что, если при выполнении программ встретятся команды SEND или RECEIVE с не определенным в системе потенциальным адресом, система SWARD не регистрирует тотчас же «ошибку», а пересылает эту команду в микропроцессоры. Если этот потенциальный адрес может быть распознан микропроцессорами (например, как соответствующий некоторому терминалу), они и выполняют указанную команду. Отдельные программы в микропроцессорах также наделены потенциальными адресами системы. Это позволяет любой программе системы взаимодействовать с ними на основе предусмотренного механизма записи и чтения. В частности,

реализованная система SWARD подсоединена к Системе 370. Если в команде RECEIVE установить необходимый потенциальный адрес, то посредством программ микропроцессора оказывается возможным получить нужный файл из Системы 370.

Возможность обращения к терминалам через потенциальные адреса как к отдельным объектам однородного множества объектов системы позволяет выйти за пределы традиционных жестких режимов работы, когда устанавливается строгая однозначная связь между пользователями терминалов и процессами. Пользователь за терминалом теперь может инициировать выполнение множества процессов и в то же время продолжать взаимодействие с операционной системой на языке ее команд. Процесс же может выполнять обмен со многими терминалами, как только ему окажутся доступными соответствующие потенциальные адреса.

УПРАЖНЕНИЯ

14.1. Каковы атрибуты и значение содержимого ячейки, представленной кодом D410237493?

14.2. Для каких целей используются ячейки «строка токенов» и «поле токенов»?

14.3. Почему с помощью ячеек «косвенный доступ к данным» и «параметр» с динамически определенным типом не могут адресоваться массивы и записи? Почему с помощью указанных ячеек нельзя обращаться также к ячейкам, тип которых определяется пользователем?

14.4. Допустим, что первая команда MOVE в программе на рис. 14.9 должна быть изменена таким образом, чтобы последовательность символов ABCDEFGH была помещена во все элементы B.N.¹⁾ Как будет в этом случае выглядеть машинная команда?

14.5. Для программы, изображенной на рис. 14.9, составьте машинную команду, обеспечивающую пересылку второго элемента массива B в первый элемент этого же массива.

14.6. Каким образом компилятор может обеспечить присвоение начальных значений элементам массива?

14.7. Какое представление в архитектуре системы SWARD получают глобальные данные (например, переменные с атрибутом EXTERNAL языка

¹⁾ То есть в первые компоненты всех записей, являющихся элементами B. — *Прим. перев.*

ПЛ/1, переменные области COMMON языка ФОРТРАН, переменные в описании пакета языка Ада)?

14.8. В каких случаях компилятор генерирует команду UNDEFINE?

14.9. Положив $CAS=IAS=3$, скомпилируйте, пользуясь карандашом и бумагой, для системы SWARD следующий модуль, написанный на языке Ада:

```
package SAMPLE is
  QUASI : array (INTEGER 1..10) of INTEGER;
  MICH  : array (INTEGER 1..10) of STRING (1..8);
end SAMPLE;
package body SAMPLE is
  procedure SAMSTORE
    (Q : in INTEGER;
     M : in STRING;
     N : in INTEGER) is
  begin
    if QUASI(N) = 0 then
      QUASI(N) := Q;
      MICH(N)(Q..Q+M'LENGTH-1) := M;
      -- store string M beginning at Qth position in MICH(N)
    else
      BUFFALO(MICH(N));
    end if ;
  end SAMSTORE;
  procedure SAMSTOP (Q : in INTEGER) is
  begin
    for I in 1..10 loop
      QUASI(I) := QUASI(I) + I * Q;
    end loop ;
  end SAMSTOP;
end SAMPLE;
```

14.10. Укажите, какие улучшения могут быть внесены в текст предыдущей программы оптимизирующим компилятором.

14.11. Предложите несколько типов команд, которые целесообразно включить в расширенное множество команд для языков КОБОЛ, ФОРТРАН, ПЛ/1, Паскаль и Ада. Не понадобятся ли для этого дополнительные типы ячеек?

ОГЛАВЛЕНИЕ

Предисловие редактора перевода	5
Предисловие ко второму изданию	7
Предисловие к первому изданию	8
Часть I. Необходимость усовершенствования архитектуры ЭВМ	10
Глава 1. Определение понятия «архитектура ЭВМ»	10
Глава 2. Критика традиционной архитектуры фон Неймана	25
Глава 3. Привязка программ к машинам	58
Глава 4. Набор средств для совершенствования архитектуры вычислительной системы	79
Часть II. Архитектура, ориентированная на язык программирования	165
Глава 5. Учебная ПЛ-машина	165
Глава 6. Трансляция и выполнение программы учебной ПЛ-машины	174
Глава 7. Набор команд учебной ПЛ-машины	190
Часть III. Архитектура машин языков программирования высокого уровня	203
Глава 8. Архитектура системы SYMBOL	203
Глава 9. Архитектура центрального процессора системы SYMBOL	218
Глава 10. Внутренняя структура и взаимосвязь процессоров системы SYMBOL	234
Часть IV. Архитектура, ориентированная на группу языков программирования	262
Глава 11. Вычислительная система B1700 фирмы Burroughs	262
Глава 12. Архитектура ЭВМ системы B1700 фирмы Burroughs, ориентированная на языки КОБОЛ и РПГ	268
Часть V. Архитектура, ориентированная на программное обеспечение ЭВМ	290
Глава 13. Назначение архитектуры системы SWARD	290
Глава 14. Система SWARD	308

Уважаемый читатель!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, изд-во «Мир».

ГЛЕНФОРД ДЖ. МАЙЕРС

АРХИТЕКТУРА СОВРЕМЕННЫХ ЭВМ

Научный редактор Л. А. Паршина
Младший научный редактор Л. В. Тарасова
Художественный редактор Н. М. Иванов
Художник В. В. Дулько
Технический редактор И. И. Володина
Корректор Н. В. Андреева

ИБ № 4067

Сдано в набор 25.07.84. Подписано в печать 28.11.84.
Формат 60×90^{1/8}. Бумага ки-жури.
Гарнитура литературная. Печать высокая.
Объем 11,50 бум. л. Усл. печ. л. 23,0.
Уч.-изд. л. 23,64. Усл. кр.-отт. 23,0.
Изд. № 20/3332. Тираж 50 000 экз.
Зак. № 283. Цена 2 руб.

Издательство «Мир»
129820, Москва, И-110, ГСП, 1-й Рижский пер., 2.

Московская типография № 11 Союзполиграфпрома
при Государственном комитете СССР
по делам издательств, полиграфии
и книжной торговли.
Москва, 113105, Нагатинская ул., д. 1.

ИЗДАТЕЛЬСТВО «МИР»

в 1983 году

выпустило книгу

МЕТОДЫ АВТОМАТИЧЕСКОГО РАСПОЗНАВА-
НИЯ РЕЧИ: В 2-х книгах. Пер. с англ./Под ред.
У. Лн. Кн. 1, 23, 98 л., цена 3 р. 40 к. Кн. 2, 27, 49 л.,
цена 3 р. 80 к.

В книге 1 рассмотрены акустические и фонетические методы распознавания речи, применяемые в настоящее время при автоматическом вводе речевой информации в автоматизированных системах проектирования, конструирования, технологической подготовки производства и управления производственными процессами. Большое внимание уделяется коррекции ошибок при распознавании речи разными методами и нахождению оптимальных способов определения правильности вводимых в ЭВМ слов, а также управляющим структурам для построения различных систем распознавания.

Предназначена для системотехников, инженеров-математиков и филологов, занимающихся автоматизацией проектирования и конструирования.

ИЗДАТЕЛЬСТВО «МИР»

в 1984 году

выпустило книгу

Р. ИЗЕРМАН. Цифровые системы управления.
Пер. с англ., 31,60 л., цена 2 р. 50 к.

В труде специалиста из ФРГ рассмотрены современные методы расчета и проектирования цифровых систем управления с детерминированными и случайными возмущениями. Значительное внимание уделено теории многосвязных и адаптивных систем управления.

Для специалистов в области автоматического управления и вычислительной техники, а также аспирантов и студентов соответствующих специальностей.

ИЗДАТЕЛЬСТВО «МИР»

в 1984 году

выпустило книгу

В. ФРИТЧ. Применение микропроцессоров в системах управления. Пер. с нем., 28,99 л., цена 2 р. 30 к.

В книге известного ученого из ГДР рассматриваются принципы построения и структурные схемы применения микропроцессоров в системах автоматического управления производственными процессами. Много внимания уделено вопросам стыковки микропроцессоров с регуляторами или системами, обеспечения надлежащего качества протекания процессов и создания наиболее экономичных структур.

Для специалистов в области автоматики и вычислительной техники, а также студентов старших курсов соответствующих специальностей вузов.

